Python: Wprowadzenie do programowania

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2024-10-03

1 Wprowadzenie

Python jest wysokopoziomowym językiem programowania ogólnego przeznaczenia. Oznacza to że jego składnia została tak zbudowana aby maksymalizować czytelność kodu dla człowieka i być niezależna od sprzętowych i implementacyjnych detali oraz że nie ma pojedynczego dedykowanego obszaru zastosowań (z łatwością może być stosowany do różnych zastosowań).

Wsórd cech i zalet Pythona należy wymienić:

- jest językiem interpretowanym[1], co daje łatwiejsze modyfikowanie kodu, eksperymentowanie z nim, itd
- jest jednym z najpopularniejszych języków programowania (wg niektórych źródeł nawet najpopularniejszym), więc nie jest ”dydaktyczną egzotyką”, która potem do niczego się nie przyda
- działa na wielu różnych platformach sprzętowych i na różnych systemach oparacyjnych
- istnieje bardzo wiele bibliotek pythonowych (posiadających pythonowe API), a można korzystać także z ”niedostosowanych” do Pythona bibliotek C (.so, .dll)
- jest łatwo rozszerzalny przy pomocy (własnych) bibliotek/modułów tworzonych w C/C++ (podstawowy interpreter napisany jest C)
- kod pythonowy może być łatwo wywoływany z poziomu C/C++, co pozwala na łatwe wykorzystanie Pythona jako języka skryptowego dla projektów tworzonych w C/C++

Ponadto Python jest wygodniejszy w uczeniu od wielu innych języków m.in. ze względu na to że kod pythonowy realizujący tą samą funkcjonalność, przy takim samym poziomie obsługi błędów, etc i podobnej czytelności, praktycznie zawsze jest krótszy od kodu C (a potrafi być krótszy kilkukrotnie, więc łatwiej go pokazać i omówić).

W ramach kursu zajmować się będziemy językiem Python w wersji 3 (czyli o pełnym numerze zaczynającym się od 3, np. 3.7.1). Należy o tym pamiętać i zwracać na to uwagę, gdyż wersja ta różni się na tyle znacząco w stosunku co do starszej, lecz wciąż używanej wersji 2, że programy, prezentowane w tym skrypcie i te które będziemy pisać na zajęciach nie będą działać w drugiej wersji Pythona.

1. może być i w niektórych sytuacjach podlega kompilacji do kodu pośredniego celem zwiększenia wydajności

Informacja

Często w zadaniach programistycznych i zawsze w ramach tego kursu jeżeli jest powiedziane:

- „napisz funkcję" to znaczy że ma zostać napisana funkcja, a nie jedynie kod programu, który mógłby stanowić wnętrze (ciało) tej funkcji,
- „napisz program" to znaczy że ma zostać napisany pełny kod programu realizujący podane czynności,
- „napisz pętlę/warunek/..." to znaczy że wystarczy napisać sam kod pętli, warunku, innej konstrukcji (ale nie tylko jego wnętrze, lecz kod całej żądanej konstrukcji składniowej),
- „napisz funkcję przyjmującą napis" to znaczy że funkcja ma mieć argument, który będzie traktowany jako napis (nie oznacza to że wymaga się wczytania tego napisu „z klawiatury"[a]),
- „napisz funkcję zwracającą X" to znaczy że funkcja ma zwrócić (poprzez return) wartość określoną przez X (nie ma jej wypisywać na ekran),
- „napisz funkcję wypisującą X" to znaczy że funkcja ma wypisać na ekran (standardowe wyjście) wartość określoną przez X,

a. Powszechnie używane w nauce programowania wczytywanie danych „z klawiatury" / odpytywanie użytkownika o kolejne parametry na ogół nie jest najlepszym rozwiązaniem programistycznym, o tym dlaczego dowiesz się w dalszych częściach tego kursu

1.1 Praca z konsolą interaktywną

Pierwszym sposobem pracy z Pythonem jest praca w interaktywnej konsoli. Uzyskujemy ją po uruchomieniu polecenia `python3`. W konsoli tej początkowo wypisane są pewne informacje (m.in. używana wersja Pythona) oraz znak zachęty (w Pythonie najczęściej >>>)[2]. Interpreter oczekuje, iż po tym znaku wpiszemy polecenie i naciśniemy Enter. Wynik polecenia zostanie wypisany w kolejnym wierszu.

Najprostszym sposobem użycia konsoli Pythona jest użycie jej jako kalkulatora — wpisujemy działanie do obliczenia, naciskamy Enter i w kolejnym wierszu otrzymujemy wynik działania. Przykład użycia konsoli Pythona jako kalkulatora znajduje się poniżej:

```
>>> 2 + 2 * 2
6
>>> (2+2) * 2
8
>>> 2 ** 7
128
>>> 47 / 10
4.7
>>> 47 // 10
4
>>> 47 % 10
7
```

W powyższym przykładzie:

- Znak ** oznacza podnoszenie do potęgi.
- Znak / oznacza dzielenie.
- Znak // oznacza dzielenie całkowite.
- Znak % oznacza branie reszty z dzielenia.
- Nawiasy okrągłe służą grupowaniu wyrażeń i wymuszaniu innej niż standardowa kolejności działań.
- Spacje nie mają znaczenia (używamy ich jedynie dla zwiększenia czytelności).

2. Zauważ że jest on inny niż znak zachęty bash'a (zazwyczaj $ poprzedzony dodatkowymi informacjami) – pozwala to na identyfikację interpretera poleceń w którym aktualnie pracujemy i wydawanie w odpowiedniej składni (bash nie rozumie poleceń w składni pythona, python nie rozumie poleceń w składni basha).

Porada

W konsoli interaktywnej przy pomocy strzałek góra/dół można przeglądać historię wydanych poleceń. Polecenia te można także wykonać ponownie (naciskając enter), a przedtem także zmodyfikować (poruszając się strzałkami prawo lewo).
Konsola ta posiada także mechanizm dopełniania wpisywanych poleceń przy pomocy tabulatora (pojedyncze naciśnięcie dopełnia, gdy tylko jedna propozycja, podwójne wyświetla propozycje dopełnień).

1.1.1 Zmienne

Podobnie jak w kalkulatorze możemy korzystać z *pamięci*, w Pythonie możemy zapisywać wartości w *zmiennych*:

```
>>> x = 3
>>> y = 4
>>> x
3
>>> x**2 + y**2
25
```

W pierwszych dwóch linijkach następuje *przypisanie* wartości 3 do zmiennej `x` oraz wartości 4 do zmiennej y. Od tej pory możemy korzystać z tych zmiennych, np. do obliczenia wartości wyrażenia $(x^2 + y^2)$.

1.1.2 Moduły i zaawansowany kalkulator

Python pozwala na wykonywanie bardziej zaawansowanych obliczeń. Możliwe jest m.in. obliczenia wartości wyrażeń logicznych, konwertowanie systemów liczbowych, obliczanie wartości funkcji trygonometrycznych. Duża część funkcji matematycznych w Pythonie zawarta jest w module „math", który wymaga zaimportowania. Można to zrobić na przykład w sposób następujący:

```
>>> import math
>>> math.sin(math.pi/2)
1.0
```

Zauważ, że odwołanie do elementów tak zaimportowanego modułu wymaga podania jego nazwy, następnie kropki i nazwy używanej funkcji z tego modułu.

1.2 Pisanie i uruchamianie kodu programu

Do tej pory korzystaliśmy z Pythona używając interaktywnej konsoli. Jest to całkiem wygodne narzędzie, jeśli wykonujemy tylko jednolinijkowe polecenia, jednak pisanie dłuższych fragmentów kodu w tej konsoli staje się już bardzo niewygodne. Drugą metodą korzystania z Pythona jest pisanie kodu programu (skryptu) w pliku tekstowym i uruchamianie tego kodu w konsoli.

Moduły

Nazwa pliku powinna być inna niż nazwy importowanych modułów, czyli jeżeli w kodzie mamy `import abc` to nasz plik nie powinien nazywać się `abc.py`, w przeciwnym razie zamiast wskazanego modułu Python będzie próbował zaimportować nasz plik.

Utwórz plik `mojProgram.py`[3] z następującą zawartością:

3. Pliki z skryptami Pythona tradycyjnie mają rozszerzenie `.py`. Nie jest ono jednak wymagane — interpreter Pythona wykona kod z pliku o dowolnym rozszerzeniu a także z pliku bez rozszerzenia.

```
x = 3
y = 4
print(x**2 + y**2)
```

W celu wykonania kodu zapisanego w pliku uruchom interpreter Pythona z jednym argumentem, będącym nazwą tego pliku: `python3 mojProgram.py`.

Porada

Zachowuj pliki z programami pisanymi w trakcie zajęć, używając nazw które pozwolą Ci łatwo zidentyfikować dany program. Mogą one być pomocne w rozwiązywaniu kolejnych zadań oraz prac domowych.

1.2.1 funkcja `print`

Zwróć uwagę, iż do wypisania wyniku działania na ekran została użyta funkcja `print`. Nie korzystaliśmy z niej wcześniej, ponieważ bazowaliśmy na domyślnym zachowaniu interpretera przy pracy interaktywnej powodującym wypisywanie na konsolę wyniku nie zapisywanego do zmiennej. Jednak kiedy tworzymy program powinniśmy w jawny sposób określać co chcemy aby zostało wypisane na konsolę właśnie np. za pomocą funkcji `print`.

Funkcja `print` wypisuje przekazane do niej (rozdzielane przecinkami) argumenty rozdzielając je spacjami. Przechodzi ona domyślnie do następnej linii po każdym wywołaniu. Na przykład:

```
print("raz dwa", "trzy ...")
print(4, 5)
```

```
raz dwa trzy ...
4 5
```

Informacja

Ilekroć w niniejszych materiałach pojawią się dwie ramki, jedna obok drugiej, w lewej ramce znajdował się będzie kod programu, a w prawej efekt jego działania wyświetlony w konsoli:

Zachowanie funkcji `print` można zmienić, dodając do jej wywołania, na końcu listy argumentów argument postaci `end = X` i/lub `sep = Y`, gdzie `X` to otoczony apostrofami ciąg znaków, który chcemy wypisywać zamiast przejścia do nowej linii, a `Y` to otoczony apostrofami ciąg znaków, który chcemy wypisywać zamiast spacji rozdzielającej wypisania kolejnych argumentów. Na przykład:

```
x = 3
y = 4
print(x, '+ ', end='')
print(y, x + y, sep=' = ')
```

```
3 + 4 = 7
```

Napisy

Ciąg znaków ujęty w apostrofy lub cudzysłowy (w Pythonie nie ma znaczenia, której wersji użyjemy, ważne jest tylko aby znak rozpoczynający i kończący był taki sam) nazywamy napisem. Możemy ich używać nie tylko w ramach funkcji print, ale też np. przypisywać do zmiennych. Więcej o napisach dowiemy się później.

1.2.2 Komentarze

Często chcemy móc umieścić w kodzie programu dodatkową informację, która ułatwi nam jego czytanie i zrozumienie w przyszłości. Służą do tego tak zwane komentarze, które są ignorowane przez interpreter (bądź kompilator) danego języka. W Pythonie podstawowym typem komentarza, jest komentarz jednoliniowy, rozpoczynający się od znaku # a kończący z końcem linii.

1.2.3 inne sposoby uruchamiania kodu z pliku 🤓

Jeżeli do wywołania interpretera Pythona dodamy opcję `-i` (np. `python3 -i mojProgram.py`) po wykonaniu kodu z podanego pliku uruchomi on konsolę interaktywną w której będą dostępne elementy (m.in. zmienne) zdefiniowane w podanym pliku.

Możliwe jest także włączenie kodu z pliku do aktualnie uruchomionego interpretera (np. konsoli interaktywnej), w taki sposób jakbyśmy go wpisali (czyli z wykonaniem wszystkich instrukcji i późniejszą możliwością dostępu do zdefiniowanych tam elementów). Aby wczytać w ten sposób kod z pliku `mojProgram.py` należy wykonać: `exec(open('mojProgram.py').read())`

1.2.4 ipython 🤓

`ipython3` jest wygodniejszym w pracy interaktywnej interpreterem Pythona w wersji 3. Pozwala on m.in. na lepsze przewijanie i edytowanie poleceń wieloliniowych w historii.

2 Podstawowe elementy składniowe

2.1 Definiowanie własnych funkcji

Bardzo często będziemy chcieli móc wielokrotnie wykorzystać raz napisany fragment kodu. W tym celu będziemy tworzyć własne *funkcje*. Definicja funkcji ma następującą postać:

```
def nazwa_funkcji(argumenty):
  pierwsze_polecenie
  drugie_polecenie
  ...
```

Zwróć uwagę na kilka rzeczy:

- Na końcu pierwszej linijki jest dwukropek.
- Druga linijka musi być *wcięta*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji.
- Jeżeli w ramach funkcji chcemy wykonać kilka instrukcji muszą one mieć taki sam poziom wcięcia.
- „Wnętrze" funkcji kończymy wracając do takiego samego poziomu wcięcia na jakim ją rozpoczęliśmy (takiego wcięcia jakie miała linijka z słowem kluczowym `def`).

Jest to typowy sposób wyznaczania bloku kodu w Pythonie i będziemy go jeszcze spotykać w innych konstrukcjach (które poznamy już niedługo), dlatego szczególnie wart jest zapamiętania.

Gdy umieszczamy inną konstrukcję korzystającą z bloku kodu we wnętrzu jakiegoś innego bloku (np. funkcji), blok tej instrukcji musi być „bardziej" wcięty od bloku w którym jest zawarty, powrót do poziomu wcięcia zewnętrznego bloku oznacza zakończenie bloku tej instrukcji i kontynuowanie zewnętrznego bloku.

Porada

Na funkcję można patrzeć jak na nazwany kawałek kodu, który możemy wywołać z innego miejsca ze odmiennymi wartościami zmiennych stanowiących jej argumenty.

Polecenie wywołania funkcji ma postać `nazwa_funkcji(argumenty)` i możemy napisać je w tym samym pliku, poniżej definicji tej funkcji. Typowo ilość i kolejność argumentów w definicji, jak i w wywołaniu powinny być takie same. Jeżeli nasza funkcja nie potrzebuje przyjmować argumentów nawiasy okrągłe w jej definicji i wywołaniu pozostawiamy puste. Jeżeli potrzebujemy więcej argumentów rozdzielamy je w obu przypadkach przecinkami (tak jak miało to miejsce w korzystaniu z funkcji `print`).

Przykład Napiszmy funkcję, która wypisuje swój argument podniesiony do kwadratu i wywołajmy ją:

```
def kwadrat(x):
  print(x * x)

kwadrat(7)
kwadrat(2 + 3)
```

```
49
25
```

Zwróć uwagę, iż wywołania funkcji w powyższym przykładzie nie są wcięte — są poza blokiem funkcji.

Polecenia wieloliniowe w konsoli interaktywnej

Możliwe jest wprowadzanie poleceń wieloliniowych w konsoli interaktywnej. W takim wypadku po wprowadzeniu pierwszej linii (rozpoczynającej blok, np. `def`) nastąpi zmiana znaku zachęty na ..., co oznacza tryb wprowadzania bloku poleceń. Następnie wprowadzamy kolejne instrukcje wykonywane w ramach tego bloku (np. funkcji) pamiętając o wcięciach. Wprowadzanie bloku kończymy pustą linią, po czym znak zachęty powróci do standardowego >>>.

Dla znających C lub C++ (1/2)

- kolejne instrukcje (zamiast średnika) kończy znak nowej linii
- średnik na końcu instrukcji (linii) nie jest błędem składniowym (jest ignorowany)
- bloki rozpoczyna dwukropek, a wyznacza je wcięcie o danej ilości znaków (nie mieszamy tabulatorów z spacjami)

2.1.1 Wartość zwracana z funkcji

Często chcemy aby funkcja zamiast wypisać wynik swojego działania na ekran zwróciła go w taki sposób aby można było go zapisać do jakiejś zmiennej, możliwe to jest poprzez zastosowanie instrukcji **return**. Przerywa ona działanie funkcji w miejscu w którym została wykonała, powoduje powrót do miejsca gdzie wywołana została funkcja i zwraca podaną do niej wartość:

```
def kwadrat(x):
  return x * x

a = kwadrat(7)
print( a - 2, kwadrat(4) )
```

```
47 16
```

2.1.2 Argumenty domyślne i nazwane

Możliwe jest podanie wartości domyślnych dla wybranych argumentów funkcji. Utworzy to z nich argumenty opcjonalne, które nie muszą być podawane przy wywołaniu funkcji. Argumenty z wartościami domyślnymi muszą występować w definicji funkcji po argumentach bez takich wartości. Przy wywołaniu funkcji można odwoływać się do jej argumentów z podaniem ich nazw, pozwala to na podawanie argumentów w innej kolejności niż podana w definicji funkcji, co jest przydatne zwłaszcza przy funkcjach z wieloma argumentami opcjonalnymi.

```
def potega(a = 2, b = 2):
    return a ** b

print( potega(), potega(4), potega(4, 3) )
print( potega(b = 3), potega(b = 1, a = 4) )
```

```
4 16 64
8 4
```

2.1.3 Zasięg zmiennej

W Pythonie wewnątrz funkcji widoczne są zmienne zdefiniowane poza nią, jednak aby móc modyfikować taką zmienną wewnątrz funkcji należy ją tam zadeklarować jako globalną przy pomocy słowa kluczowego **global**:

```
def test():
  global b
  a, b = 5, 13
  print(a, b, c)

a, b, c = 1, 3, 7
test()
print(a, b, c)
```

```
5 13 7
1 13 7
```

Analizując działanie powyższego kodu zwrócić uwagę na:

- zasłonięcie globalnego `a` poprzez lokalne a wewnątrz funkcji (nie można zmodyfikować globalnej zmiennej `a` w funkcji),
- możliwość dostępu do globalnych zmiennych w funkcji dopóki ich nie zasłonimy zmienną lokalną (tak używamy zmiennej `c`)
- możliwość zmodyfikowania zmiennej globalnej gdy jest zadeklarowana w funkcji jako **`global`**

Zadanie 2.1.1

Napisz funkcję, która przyjmuje dwa argumenty i zwraca ich sumę. Użyj jej do obliczenia (oraz wypisania na konsolę) wartości kilku różnych sum.
Wskazówka: `print()` powinien być użyty na zewnątrz tej funkcji.

2.2 Pętla `for`

Załóżmy, że chcemy obliczyć kwadraty wszystkich liczb od 1 do 4. Zgodnie z dotychczasową wiedzą, w tym celu musimy wykonać 4 działania:

```
print(1 * 1)
print(2 * 2)
print(3 * 3)
print(4 * 4)
```

```
1
4
9
16
```

Widzimy jednak, że te działania są bardzo podobne i chciałoby się je wykonać „za jednym zamachem". Do wykonywania wielokrotnie tego samego (lub podobnego) kodu służą pętle. Najprostszym rodzajem pętli jest pętla **`for`**, która dla danej *listy* i operacji do wykonania wykonuje tę operację po kolei na każdym elemencie listy.

Do wykonania powyższego zadania służy pętla **`for`** w następującej postaci:

```
for x in [1, 2, 3, 4]:
    print(x * x)
```

```
1
4
9
16
```

Spróbuj przepisać tę pętlę i uruchomić program. Zauważ że wnętrze pętli jest wyznaczone w sposób analogiczny do wnętrza funkcji:

- Rozpoczyna się od dwukropka kończącego pierwszą linię.
- Kolejne linijki są *wcięte*, tzn. rozpoczynać się od spacji, kilku spacji lub znaku tabulacji.
- Jeżeli w ramach pętli chcielibyśmy wykonać kilka instrukcji muszą one mieć taki sam poziom wcięcia.
- „Wnętrze" pętli kończymy wracając do takiego samego poziomu wcięcia na jakim ją rozpoczęliśmy (takiego wcięcia jakie miała linijka z słowem kluczowym **`for`**).
- Pętle możemy zagnieżdżać jedna w drugiej — blok wewnętrznej pętli musi być „bardziej" wcięty. Powrót do poziomu wcięcia zewnętrznej pętli oznacza zakończenie pętli wewnętrznej i kontynuowanie zewnętrznej.

2.3 Lista kolejnych liczb naturalnych

Często potrzebujemy, aby pętla przeszła po liście kilku kolejnych liczb naturalnych. W tym celu możemy oczywiście podać wprost kolejne elementy listy (tak jak w powyższym przykładzie), jednak istnieje wygodniejsze rozwiązanie, mianowicie polecenie `range()`:

```
for x in range(7):
    print(x, end = ', ')
```

```
0, 1, 2, 3, 4, 5, 6,
```

```
for x in range(5, 10):
    print(x, end = ', ')
```

```
5, 6, 7, 8, 9,
```

```
for x in range(10, 20, 3):
    print(x, end = ', ')
```

```
10, 13, 16, 19,
```

Na powyższych przykładach widzimy, że polecenie `range()` występuje w trzech wersjach:

- `range(kon)` generuje listę kolejnych liczb od 0 (**włącznie**) do `kon` (**wyłącznie**).
- `range(pocz, kon)` generuje listę kolejnych liczb od `pocz` (**włącznie**) do `kon` (**wyłącznie**).
- `range(pocz, kon, krok)` generuje listę liczb od `pocz` (**włącznie**) do `kon` (**wyłącznie**), przeskakując w każdym kroku o `krok`.

Do zapamiętania

Wszystkie przedziały w Pythonie są domknięte z lewej strony i otwarte z prawej strony, tzn. zawierają swój lewy koniec i nie zawierają swojego prawego końca.

Zadanie 2.3.1

Napisz program obliczający sumę $1^2 + 2^2 + 3^2 + ... + 99^2 + 100^2$.

2.4 Typ logiczny

Jak już się przekonaliśmy można używać Pythona jako kalkulatora. Możemy go także użyć do obliczania wartości wyrażeń logicznych. Służy do tego wbudowany dwuwartościowy typ logiczny z wartościami:

- **`True`** oznaczającą logiczną jedynkę / prawdę
- **`False`** oznaczającą logiczne zero / fałsz

Operacje na tym typie wykonujemy z użyciem słów kluczowych: **`and`**, **`or`**, **`not`** oznaczających odpowiednio: iloczyn logiczny (aby był prawdą oba warunki muszą być spełnione), sumę logiczną (aby wynik był prawdą co najmniej jednej z warunków musi być spełniony) oraz negację logiczną. Podobnie jak w zwykłych operacjach arytmetycznych możemy grupować ich fragmenty (celem wymuszenia kolejności działań) przy pomocy nawiasów okrągłych.

Wartościom tego typu mogą odpowiadać wybrane wartości innych typów (np. liczba całkowita 0 odpowiada **`False`**, a pozostałe liczby całkowite **`True`**). Wartościami tego typu są też wyniki różnego rodzaju porównań, takich jak: `<` (mniejsze), `>` (większe), `<=` (mniejsze równe), `>=` (większe równe), `==` (równe), `!=` (nierówne).

2.5 Instrukcja warunkowa `if`

Często chcemy, aby program zachowywał się w różny sposób w zależności od tego, czy jakiś warunek jest spełniony, czy nie. W Pythonie (jak w większości języków programowania) służy do tego instrukcja warunkowa **`if`**.

Przypuśćmy, że chcemy napisać funkcję, która dla podanej wartości sprawdzi czy odpowiada ona logicznej prawdzie czy fałszowi i wypisuje odpowiedni komunikat. Zatem kod będzie wyglądał następująco:

```
def sprawdz(x):
    if x:
        print(x, '-- prawda')
    else:
        print(x, '-- nie prawda')
sprawdz(1)
sprawdz(0)
```

```
1 -- prawda
0 -- nie prawda
```

Zwróć uwagę na następujące rzeczy:

- **if** to po polsku „jeśli”, **else** to po polsku „w przeciwnym przypadku”.
- Linijki rozpoczynające się od **if** i **else** (podobnie jak linijki rozpoczynające się np. od **def**) kończą się dwukropkiem.
- „Wnętrze” **if**-a i **else**-a (linijki 3 i 5) jest wcięte (bardziej niż samo wnętrze definicji funkcji sprawdz).
- Linijka 3 zostanie wykonana, jeśli spełniony będzie warunek z linijki 2, czyli jeśli wartość zmiennej x będzie odpowiadała prawdzie.
- Linijka 5 zostanie wykonana, jeśli warunek z linijki 2 nie będzie spełniony.

W powyższym przykładzie użyliśmy konstrukcji **if**/**else** do rozróżnienia pomiędzy dwoma przypadkami. Używając komendy **elif** (skrót od **else if**) możemy stworzyć bardziej skomplikowany kod do rozróżnienia pomiędzy kilkoma różnymi przypadkami:

```
for x in range(0, 5):
    if x < 1 or x == 4:
        print('mniejsze od 1 lub rowne 4')
    elif x in [0,2,3]:
        print('0 2 lub 3')
    else:
        print('nic ciekawego')
```

```
mniejsze od 1 lub rowne 4
nic ciekawego
0 2 lub 3
0 2 lub 3
mniejsze od 1 lub rowne 4
```

Ten kod składa się z trzech bloków, które są wykonywane w zależności od spełnienia poszczególnych warunków: **if**, **elif**, **else**. Mamy dużą dowolność w konstruowaniu tego typu fragmentów kodu: bloków **elif** może być dowolnie wiele, blok **else** może występować jako ostatni blok, ale może też go nie być w ogóle.

W powyższym przykładzie widzimy również, że w roli warunków sprawdzanych w ramach **if**a mogą występować bardziej złożone wyrażenia. Możemy tutaj użyć dowolnego wyrażenia którego wynik odpowiada wartości logicznej **True**/**False**, najczęściej spotkamy się z wyrażeniami złożonymi z poznanych już operatorów porównań (<, >, <=, >=, ==, !=) i operacji logicznych (**and**, **or**, **not**).

Zwróć uwagę na warunek postaci „A **in** B”. Taki warunek sprawdza, czy wartość reprezentowana przez A jest elementem B, a jego wynik oczywiście także jest wartością logiczną. W naszym przykładzie sprawdzaliśmy, czy wartość zmiennej x występuje w podanej liście liczb, czyli czy jest 1, 2 lub 3.

Zauważ, że dla x wynoszącego 0 spełnione są dwa warunki (pierwszy i środkowy), w takim wypadku decydująca jest kolejność warunków i w konstrukcji **if**/**elif** wykonany zostanie jedynie kod związany z pierwszym pasującym warunkiem.

Zadanie 2.5.1

Napisz funkcję `znak(liczba)` która wypiszę informację o znaku podanej liczby (wyróżniając zero) i zwróci jej wartość bezwzględną. Wywołanie funkcji znak powinno wyglądać następująco:

```
a = znak(7)
b = znak(-13)
c = znak(0)
print(a, b, c)
```

```
7 jest dodatnia
-13 jest ujemna
0 to zero
7 13 0
```

Dla znających C lub C++ (2/2)

- nie ma konstrukcji i++, czy też ++i, jest za to i += 1
- warunek if'a w nawiasach nie jest błędem składniowym (ale po nawiasach musi być dwukropek)
- nie ma pętli for w stylu C („trójinstrukcyjnej"), w Pythonie pętla for zawsze iteruje po elementach jakiejś listy

2.6 Pętla `while`

Do tej pory korzystaliśmy z pętli **for**, która pozwala na iterowanie po liście elementów. Innym istotnym rodzajem pętli jest pętla **while**, która powoduje wykonywanie zawartego w niej kodu dopóki podany warunek jest spełniony.

```
a, b = 0, 1
while a <= 20:
    print(a, end=" ")
    a, b = b, a + b
```

```
0 1 1 2 3 5 8 13
```

Zwróć uwagę, że wewnątrz pętli **while** (tak samo jak innych konstrukcji używających wciętego bloku - takich jak **for**, czy **if**) może znajdować się więcej niż jedno polecenie. Trzeba tylko pamiętać, aby wszystkie były poprzedzone takim samym wcięciem.

Pętla **while** jest też naturalnym wyborem gdy w Pythonie chcemy przechodzić przez jakiś zakres liczb z krokiem nie całkowitym (wcześniej poznana instrukcja `range`, stosowana do iterowania po zakresie liczbowym w pętli **for**, wymaga aby krok był całkowity).

Zauważ że pętla będzie się wykonywała dopóki warunek jest spełniony, zatem łatwo przy jej pomocy stworzyć pętlę nieskończoną – zarówno celowo jak i w wyniku błędu. Dlatego należy mieć na uwadze ryzyko zapętlenia (nieskończonego wykonywania takiej pętli na skutek jakiejś pomyłki).

Zadanie 2.6.1

Rozwiąż zadanie 2.3.1 stosując pętlę **while**.

2.7 `break` i `continue`

W ramach zarówno pętli for jak i while możemy użyć instrukcji:

- **break** powodującej przerwanie wykonywania pętli
- **continue** powodującej pominięcie pozostałych instrukcji w aktualnym obiegu pętli

Ich działanie może zobrazować poniższy kod:

```
for x in [1, 2, 3, 4, 5, 6]:
  print("start", x)
  if x == 2:
    continue
  if x == 4:
    break
  print("...")
```

```
start 1
...
start 2
start 3
...
start 4
```

2.8 Wielokrotne przypisanie

Zwróć uwagę w powyższym kodzie także na operację wielokrotnego przypisania postaci `a, b = x, y`. Dokonuje ona przypisania wartości x do a i y do b, przy czym wartości x i y obliczane są przed zmodyfikowaniem a i b. Pozwala to m.in. na zamianę wartości pomiędzy a i b bez stosowania zmiennej tymczasowej poprzez zapis: `a, b = b, a`. Podobnie możemy zapisywać przypisania większej ilości wartości do większej ilości zmiennych np: `a, b, c = 1, 5, 9`. Z notacji tej będziemy też często korzystać w dalszej części skryptu przy inicjalizacji zmiennych.

3 Napisy

Do tej pory używaliśmy zmiennych do przechowywania liczb i operowania na nich. Zmienne mogą również jako wartości przyjmować litery, słowa, a nawet całe zdania:

```
x = 'A'
a, b, c = 'Ala', "ma", " kota i psa"
d = """ ... a co ma ...
 "kotek"?"""
print(x, a[2])
print(c[1], c[-1], c[-3])
print(a + b)
print(3 * a)
print(a + " " + b + c + d)
```

```
A a
o a p
Alama
AlaAlaAla
Ala ma kota i psa ... a co ma ...
 "kotek"?
```

Zwróć uwagę na następujące rzeczy:

- Napisy muszą być otoczone pojedynczymi apostrofami lub podwójnym cudzysłowami (nie ma znaczenia, którą wersję wybierzemy), w przypadku napisów wieloliniowych używamy trzykrotnie apostrofu lub cudzysłowowa na początku i końcu napisu.
  Nie przypisane do żadnej zmiennej napisy wieloliniowe mogą być stosowane jako komentarze wieloliniowe.
- Przy użyciu liczby w nawiasie kwadratowym możemy poznać poszczególne litery napisu (***numeracja rozpoczyna się od 0***).
- Ujemny indeks oznacza odliczanie liter od końca napisu: ostatnia litera napisu `c` to `c[-1]`, przedostatnia to `c[-2]`, itd.
- Przy użyciu znaku dodawania możemy sklejać (*konkatenować*) napisy.
- Przy użyciu znaku gwiazdki możemy mnożyć napisy (czyli sklejać same ze sobą).

Innymi przydatnymi operacjami na napisach jest sprawdzanie długości napisu poleceniem `len()` oraz wycinanie podnapisu przy użyciu dwukropka:

```
tekst = 'Python'
dlugosc = len(tekst)
print(dlugosc, tekst[2:5], tekst[3:], tekst[:3])
```

```
6 tho hon Pyt
```

W powyższym przykładzie:

- komenda `tekst[2:5]` zwraca podnapis od znaku nr 2 (**włącznie**) do znaku nr 5 (**wyłącznie**),
- komenda `tekst[3:]` zwraca podnapis od znaku nr 3 (**włącznie**) do końca,
- komenda `tekst[:3]` zwraca podnapis od początku do znaku nr 3 (**wyłącznie**).

Podobnie jak w `range()` możemy podać trzeci argument określający krok. Pozwala to na wybieranie co n-tego znaku z napisu, zarówno zaczynając od początku jak i końca:

```
tekst = '123456789'
print(tekst[::2], tekst[1::2])
print(tekst[::-1], tekst[::-3])
print(tekst[::-1][::3], tekst[::3][::-1])
```

```
13579 2468
987654321 963
963 741
```

W powyższym przykładzie:

- komenda `tekst[::2]` zwraca co drugi znak,
- komenda `tekst[1::2]` zwraca co drugi znak od znaku nr 1,
- komenda `tekst[::-1]` zwraca napis od tyłu,

- komenda `tekst[::-3]` zwraca co 3 znak z napisu od tyłu (warto zauważyć że nie zawsze jest to równoważne wypisaniu napisu złożonego z co 3 znaku od tyłu).

Zadanie 3.0.1

Napisz funkcję, która dla danej listy słów wypisze każde słowo z listy wspak. Np. dla listy `['Ala', 'ma', 'kota']` funkcja powinna wypisać:

```
alA
am
atok
```

Wskazówka: Po elementach listy znajdującej się w zmiennej możemy iterować pętlą **`for`**, *tak jak robiliśmy to po literach napisu, czy po elementach listy liczb zapisanej bezpośrednio w konstrukcji pętli (spróbuj* **`for x in lista:`**).

3.1 Napis jako lista

Wszystkie listy, których do tej pory używaliśmy w pętli **`for`** były listami liczb. Okazuje się, że w Pythonie napisy mogą być traktowane jako lista, a dokładniej listą liter. Oznacza to, że po napisie można przejść przy użyciu pętli **`for`**, tak samo jak przechodziliśmy po liście liczb:

```
for l in 'Abc':
    print('litera', end = ' ')
    print(l)
```

```
litera A
litera b
litera c
```

3.1.1 Modyfikowalność napisów

Python pozwala odwoływać się do poszczególnych znaków w napisie jak do elementów listy, jednak nie pozwala na ich modyfikowanie:

```
s = "abcdefgh"
s[2] = "X"
print(s)
```

```
Traceback (most recent call last):
  File "python", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Zwróć uwagę na komunikat błędu, który został wyświetlony, podaje on informacji o tym co wywołało błąd (opis błędu) i w której linii programu on wystąpił. **Czytanie ze zrozumieniem komunikatów o błędach ułatwia naprawianie niedziałającego programu.**

Jeżeli zachodzi potrzeba modyfikowania napisu konkretnych znaków w napisie możemy użyć poznanej wcześniej metody uzyskiwania podnapisów:

```
s = "abcdefgh"
s = s[:2] + "X" + s[3:5] + s[6:]
print(s)
```

```
abXdegh
```

Powyższy przykład w miejsce znaku nr 2 wstawia napis "X" oraz usuwa znak nr 5 z napisu. Przy konieczności modyfikacji znak po znaku możemy użyć iteracji po napisie i budować nowy napis znak po znaku:

```
s, ns = "abcdefgh", ""
for z in s:
    if z in "cf":
        ns = ns + "X"
    else:
        ns = ns + z
print(ns)
```

```
abXdeXgh
```

Zadanie 3.1.1

Napisz funkcję `wyiksuj(napis)`, która zwróci dany `napis`, zastępując każdą małą literę przez `x` i każdą wielką literę przez `X`, natomiast resztę znaków pozostawi bez zmian. Np. dla napisu `'Python 3.6.1 (default, Dec 2015, 13:05:11)'` funkcja powinna zwrócić napis: `Xxxxxx 3.6.1 (xxxxxxx, Xxx 2015, 13:05:11)`

3.2 Obiektowość

Jak być może zauważyliśmy wszystkie podstawowe typy w Pythonie są klasami. Związane z tym jest m.in. to iż posiadają one metody służące do operowania na nich. Metodą nazywamy funkcję związaną z danym typem i wykonywaną na obiekcie tego typu. Zapisywane jest to z użyciem kropki, nazwy metody i nawiasów okrągłych które mogą zawierać dodatkowe argumenty. Na przykład: `"aącd".islower()` jest wywołaniem metody `islower` typu napisowego na napisie `"aącd"`; metoda ta sprawdza czy w podanym ciągu znaków nie występują wielkie litery.

Klasy posiadają także konstruktory, które możemy wywołać używając nazwy danej klasy jak funkcji i użyć np. do konwersji pomiędzy różnymi typami. Jak już wiemy wszystkie nazwy w pythonie żyją w jednym świecie, dotyczy to też nazw klas. Dlatego warto uważać aby nie nazywać swoich zmiennych zarówno tak jak nazywają się wbudowane funkcje, ani tak jak nazywają się wbudowane typy danych (takie jak int, bool, str, folat i tak dalej).

Opis danego typu wraz z dostępnymi metodami można obejrzeć przy pomocy polecenia `help()`, np. `help("str")`.

W przypadku napisów za pomocą metod tej klasy mamy możliwość między innymi wyszukania miejsca wystąpienia podnapisu, zamiany wielkich liter na małe i odwrotnie, etc.

Zadanie 3.2.1

Zapoznaj się z dokumentacją klasy odpowiedzialnej za napisy (`str`), zwróć szczególną uwagę na metody `split`, `find`, `replace`. Korzystając z metod klasy `str` napisz funkcję `parse` która dla napisu będącego jej argumentem wykona zamianę wszystkich ciągów "XY" na spację oraz dokona rozbicia napisu złożonego z pól rozdzielanych dwukropkiem na listę napisów odpowiadających poszczególnym polom. Funkcja powinna działać w następujący sposób:

```
 > l = parse("Ala:maXYkota:i inne:zwierzeta")
 > print(l)
['Ala', 'ma kota', 'i inne', 'zwierzeta']
```

3.3 Konwersje liczba – napis

Z punktu widzenia komputera liczba czy też element napisu, którym jest litera są pewną wartością numeryczną. Natomiast my do zapisu liczb używamy różnych systemów (np. dziesiętnego, czy też szesnastkowego). Domyślnie liczby wprowadzane do programu interpretowane są jako zapisane w systemie dziesiętnym, podobnie liczby uzyskiwane poprzez konwersję napisu przy pomocy funkcji `int()` (dokładniej jest to konstruktor typu całkowitego). Możliwe jest jednak wprowadzanie liczb zapisanych w innych systemach licz-

bowych lub konwersja z napisu zawierającego liczbę — drugi, opcjonalny argument `int()` pozwala określić podstawę systemu z którego konwertujemy, zero oznacza automatyczne wykrycie w oparciu o prefix:

```
# szesnastkowo
h1, h2, h3 = 0x1F, int("0x1F", 0), int("1F", 16)
# oktalnie
o1, o2, o3 = 0o17, int("0o17", 0), int("17", 8)
# binarnie
b1, b2, b3 = 0b101, int("0b101", 0), int("101", 2)

print("",h1,o1,b1, "\n",h2,o2,b2, "\n",h3,o3,b3)
```

```
31 15 5
31 15 5
31 15 5
```

Możliwe jest także konwertowanie wartości liczbowej na napis w określonym systemie liczbowym:

```
a, b = 3, 13
c = (a + b) * b
s = "(" + bin(a) + " + " + oct(b) + ") * " + hex(b) + " = " + str(c)
print( s )
```

3.4 Kodowania znaków

Python używa Unicode dla obsługi napisów, jednak przed przekazaniem napisu do świata zewnętrznego konieczne może być zastosowanie konwersji do określonej postaci bytowej (zastosowanie odpowiedniego kodowania). Służy do tego metoda `encode()` np.:

```
a = "ąąbcć ... ←↓→"
inUTF7 = a.encode('utf7')
inUTF8 =  a.encode() # lub a.encode('utf8')
print("'" + a + "' w UTF7 to: " + str(inUTF7) + ", w UTF8: " + str(inUTF8))
```

Zmienne typu 'bytes' oprócz przekazania na zewnątrz (np. zapisu do pliku lub wysłania przez sieć) mogą zostać także m.in. zdekodowane do napisu z użyciem metody `decode()` lub poddane dalszej konwersji np. kodowaniu base64:

```
print("zdekodowany UTF7: " + inUTF7.decode('utf7'))

import codecs
b64 = codecs.encode(inUTF8, 'base64')
print("napis w UTF8 po zakodowaniu base64 to: " + str(b64))
```

W powyższym przykładzie należy zwrócić uwagę na instrukcję **`import`**, która służy do załączania bibliotek pythonowych do naszego programu. W tym wypadku załączamy fragment standardowej biblioteki Pythona o nazwie `codecs`.

Base64 jest jednym z kodowań pozwalających na zapis danych binarnych w postaci ograniczonego zbioru znaków drukowalnych, co pozwala m.in. na osadzanie danych binarnych (np. obrazki) w plikach tekstowych (np. dokumenty html, pliki źródłowe programów).

3.4.1 Konwersja pomiędzy znakiem a jego numerem

Możliwe jest także konwertowanie pomiędzy liczbowym numerem znaku Unicode, a napisem go reprezentującym i w drugą stronę — służą do tego odpowiednio funkcje `chr()` i `ord()`. W ramach napisów można

też użyć \uNNNN lub \UNNNNNNNN (gdzie NNNN/NNNNNNNN jest cztero/ośmio[4] cyfrowym numerem znaku zapisanym szesnastkowo) lub po prostu umieścić dany znak w pliku kodowanym UTF8[5].

```
print(chr(0x221e) + " == \u221e == ∞ == \U0000221e")
print(hex(ord("∞")), hex(ord("\u221e")), hex(ord(chr(0x221e))) )
```

Zwróć uwagę na równoważność poszczególnych zapisów – wypisują taki sam znak na konsoli, zwracają taki sam numer unicodowy.

Niektóre znaki specjalne jak np. znak nowej linii, tabulator możemy wprowadzić z użyciem krótszych i łatwiejszych do zapamietania sekwencji niż opartych o ich numer. Dla znaku nowej linii jest to \n, a tabulatora \t.

Zadanie 3.4.1

Napisz program dekodujący napis kodowany w UTF8 zakodowany przy pomocy base64 mający postać: `b'UHl0aG9uIGplc3QgZmFqbnkg8J+Yjg==\n'`.
Wskazówka: dane wejściowe funkcji `decode()` muszą być typu "bytes", można to uzyskać poprzedzając napis prefiksem `b`, tak jak powyżej.

3.5 Wyrażenia regularne

W przetwarzaniu napisów bardzo często stosowane są wyrażenia regularne służące do dopasowywania napisów do wzorca który opisują, wyszukiwaniu/zastępowaniu tego wzorca. Do typowej, podstawowej składni wyrażeń regularnych zalicza się m.in. następujące operatory:

```
.       - dowolny znak
[a-z]   - znak z zakresu
[^a-z]  - znak z poza zakresu (aby mieć zakres z ^ należy dać go nie na początku)
^       - początek napisu/linii
$       - koniec napisu/linii

*       - dowolna ilość powtórzeń
?       - 0 lub jedno powtórzenie
+       - jedno lub więcej powtórzeń
{n,m}   - od n do m powtórzeń

()      - pod-wyrażenie (może być używane dla operatorów powtórzeń,
          a także dla referencji wstecznych)
|       - alternatywa: wystąpienie wyrażenia podanego po lewej stronie
          albo wyrażenia podanego prawej stronie
```

Python umożliwia korzystanie z wyrażeń regularnych za pomocą modułu `re`:

4. Użycie wariantu cztero cyfrowego jest możliwe jedynie dla znaków unicode o numerach mniejszych niż 0xffff
5. Użyty w przykładzie symbol nieskończoności można uzyskać na standardowej polskiej klawiaturze pod Linuxem przy pomocy kombinacji AltGr + Shift + M

```
import re
y = "aa bb cc bb ff bb ee"
x = "aa bb cc dd ff gg ee"

if re.search("[dz]", y):
  print("y zawiera d lub z")

if re.search(".*[dz]", x):
  print("x zawiera d lub z")

if re.search(" ([a-z]{2}) .* \\1", y):
  print("y zawiera dwa razy to samo")

if re.search(" ([a-z]{2}) .* \\1", x):
  print("x zawiera dwa razy to samo")
```

```
x zawiera d lub z
y zawiera dwa razy to samo
```

Funkcja search zwraca więcej informacji niż sam fakt pasowania lub nie pasowania:

```
import re
x = "aa bb cc dd ff gg ee"

# wypisanie dopasowania
wynik = re.search("cc (xx)|(dd) ff", x)
if wynik:
  print( "dopasowano tekst:", wynik.group(0) )
  print( "na pozycji:", wynik.span()[0] )

wynik = re.search("cc (xx|dd) ff", x)
if wynik:
  print( "dopasowano tekst:", wynik.group(0) )
  print( "na pozycji:", wynik.span()[0] )
```

```
dopasowano tekst: dd ff
na pozycji: 9
dopasowano tekst: cc dd ff
na pozycji: 6
```

Wyrażeń regularnych możemy używać także do operacji wyszukaj i zastąp pasujący fragment napisu:

```
import re
y = "aa bb cc bb ff bb ee"

# zastępowanie
print (re.sub('[bc]+', "XX", y, 2))
print (re.sub('[bc]+', "XX", y))

# zachłanność
print (re.sub('bb (.*) bb', "X \\1 X", y))
print (re.sub('.*bb (.*) bb.*', "\\1", y))
print (re.sub('.*?bb (.*) bb.*', "\\1", y))
```

```
aa XX XX bb ff bb ee
aa XX XX XX ff XX ee
aa X cc bb ff X ee
ff
cc bb ff
```

Zwróć uwagę na:

- Działanie funkcji search, która wyszukuje podnapis pasujący do wyrażenia i umożliwia zaróno uzyskanie pasującego podnapisu, jak też samej informacji o fakcie pasowania lub nie do wyrażenia.
- Działanie alternatywy i nawiasów - standardowo alternatywa obejmuje wszystko co po lewej kontra wszystko co po prawej, nawiasy obejmujące fragment prawej bądź lewej strony na to nie wpływają

(`cc (xx)|(dd) ff` nie zadziało jako "xx" ablo "dd" pomiędzy "cc" a "ff", a jako "cc xx" albo "dd ff"), aby ograniczyć działanie alternatywy tylko do fragmentu wyrażenia należy objąć nawiasami ten fragment wraz z alternatywą w nim umieszczoną (`cc (xx|dd) ff` zadziało jako "xx" ablo "dd" pomiędzy "cc" a "ff".

- Odwołania wsteczne do pod-wyrażeń (fragmentów ujętych w nawiasy) postaci `\\x`, gdzie `x` jest numerem pod-wyrażenia.
- „*Zachłanność*" (ang. *greedy*) wyrażeń regularnych:
  - w pierwszym wypadku `bb (.*) bb` dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`,
  - w drugim przypadku gdy zostało poprzedzone `.*` dopasowało tylko `ff`, gdyż `.*` dopasowało najdłuższy możliwy fragment czyli `aa bb cc`,
  - w trzecim wypadku `bb (.*) bb` mogło i dopasowało najdłuższy możliwy fragment, czyli `cc bb ff`, gdyż było poprzedzone niezachłanną odmianą dopasowania dowolnego napisu, czyli: `.*?`.

  Po każdym z operatorów powtórzeń (`. ? + {n,m}`) możemy dodać pytajnik (`.? ?? +? {n,m}?`) aby wskazać że ma on dopasowywać najmniejszy możliwy fragment, czyli ma działać nie zachłannie.

Zadanie 3.5.1

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest słowem (tzn. nie zawiera spacji).

Zadanie 3.5.2

Napisz funkcję która sprawdzi z użyciem wyrażeń regularnych czy dany napis jest liczbą (tzn. jest złożony z cyfr i kropki, a na początku może wystąpić + albo -).

Argumenty linii poleceń

Tworzone przez nas skrypty pythonowe mogą przyjmować (tak jak różne inne poznane programy) argumenty (i opcje) w linii poleceń. Aby mieć dostęp do listy argumentów przekazanych w linii poleceń należy skorzystać z `sys.argv`:

```
import sys
print(sys.argv)
```

```
$ python3 /tmp/skrypt.py aa tt
['/tmp/skrypt.py', 'aa', 'tt']
```

Jak możemy zauważyć jest to standardowa pythonowa lista zawierająca w kolejnych swoich elementach nazwę z którą został wywołany nasz skrypt i kolejne przekazane do niego argumenty.
Moduł argparse dostarcza dość wygodny parser opcji w standardowej notacji (długie z dwoma myślnikami, krótkie z jednym, itd.):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument(
  '-v', "--verbose", action="store_true",
  help='opcja typu przełącznik'
)
parser.add_argument(
  'ARG', nargs='?',
  help='argument pozycyjny (opcjonalny)'
)
args = parser.parse_args()
print(args)
```

```
$ python3 /tmp/skrypt.py aa tt
usage: skrypt.py [-h] [-v] [ARG]
skrypt.py: error: unrecognized
 ↪ arguments: tt
$ python3 /tmp/skrypt.py -v aa
Namespace(ARG='aa', verbose=True)
```

To tylko mały przykład możliwości argparse, a szczegóły (jak zwykle) – w dokumentacji.

4 Zmienne i ich typy

4.1 Określanie typu zmiennej

Do tej pory poznaliśmy kilka typów zmiennych w Pythonie: liczby, napisy oraz listy. Poznaliśmy także metody konwersji pomiędzy niektórymi z typów (np. instrukcje `str()`, `int()`). Jeżeli chcemy dowiedzieć się jakiego typu jest dana zmienna możemy skorzystać z funkcji `type()`:

```
a, b, c = 1, 3.14, "Python"
print(a, type(a))
print(b, type(b))
print(c, type(c))
c = (a == 1)
print(c, type(c))
```

```
1 <class 'int'>
3.14 <class 'float'>
Python <class 'str'>
True <class 'bool'>
```

Zauważ że inny typ związany jest z liczbami całkowitymi, inny z rzeczywistymi, a jeszcze inny z wartościami logicznymi (**`True`**/**`False`**). Zauważ także, że zmienna może zmienić swój typ.

4.1.1 Typowanie w Pythonie a w innych językach 😳

Typowanie, czyli określanie typu zmiennej, w Pythonie można porównać do typowania w współczesnym C++ z użyciem słowa kluczowego auto. Python:

- określa zmiennej w momencie napotkania jej deklaracji na podstawie wartości do niej przypisywanej (tak samo jak C++ robi dla zmiennych auto)

```
#include <stdio.h>
int main() {
  auto a = 1;
  printf("%d", a);
}
```

- nie pozwala odwołać się do zmiennej nie zadeklarowanej (np. PHP pozwala, generując jedynie "Notice")

```
<?php
$a = $b +1;
echo $a, $b;
?>
```

- pozwala na zmianę typu zmiennej w trakcie działania (C++ nie pozwala nawet z typem auto)

```
a = "abc"
print(a, type(a))
a = 1
print(a, type(a))
```

4.1.2 Wielkość zmiennej typu int 😳

Python nie posiada wbudowanego ograniczania wielkości liczb całkowitych, jednak wielkość wartości przechowywanej w tym typie może mieć wpływ na rozmiar zmiennej.

```
x = 1
print(x, type(x), x.__sizeof__())
x = 12**10
print(x, type(x), x.__sizeof__())
x = 12**20
print(x, type(x), x.__sizeof__())
x = 13
print(x, type(x), x.__sizeof__())
```

```
1 <class 'int'> 28
61917364224 <class 'int'> 32
383375992447475122176 <class 'int'> 36
13 <class 'int'> 28
```

4.2 Listy

Do tej pory listy traktowaliśmy głównie jako zbiór elementów po którym iterujemy. Zastosowanie list jest jednak znacznie szersze. Lista stanowi pewnego rodzaju kontener do przechowywania innych zmiennych, w którym elementy zorganizowane są na zasadzie określenia ich (względnej) kolejności. Lista może zawierać elementy różnych typów.

Na listach możemy wykonywać m.in. operacje modyfikowania, czy też usuwania jej elementów:

```
l = ["i", "C", 0, "M"]
l[0] = "I"
del l[2]
print(l)
```

```
['I', 'C', 'M']
```

W powyższym przykładzie widzimy:

- Modyfikację pierwszego elementu listy (`l[0] = "I"`), z użyciem odwołania poprzez numer elementu. Elementy list numerujemy od zera. Ujemne wartości oznaczają numerowanie od końca listy, czyli -1 jest ostatnim elementem listy, -2 przedostatnim, itd.
- Usunięcie trzeciego elementu listy (`del l[2]`). Powoduje to zmianę numeracji kolejnych elementów.

Jednak jeżeli chcemy modyfikować elementy listy iterując po niej, to konieczne jest iterowanie po indeksach (a nie jak dotychczas po wartościach):

```
for i in range(len(l)):
    print(l[i])
    l[i] = "q"
print(l)
```

```
I
C
M
['q', 'q', 'q']
```

Dzieje się tak gdyż przypisanie do zmiennej x jakiejś wartości w ramach konstrukcji `for x in lista:` modyfikuje tylko zmienną x, a nie element listy który został do niej pobrany.

4.2.1 Wybór podlisty

Możemy także tworzyć „podlisty” przy pomocy operatora zakresów w identyczny sposób jak to zostało opisane przy napisach, np. `ll[1::2]` zwróci listę złożoną z co drugiego elementu listy `ll` zaczynając od elementu o indeksie 1.

4.2.2 Lista jako modyfikowalny napis

Listy mogą też służyć jako narzędzie do modyfikowania napisów. W tym celu można skorzystać np. z listy złożonej z liter oryginalnego napisu:

```
s = "abcdefgh"
l = list(s)
l[2] = "X"
del(l[5])
s = str.join("", l)
print(s)
```

```
abXdegh
```

4.2.3 Obiektowość list

W przypadku list za pomocą metod tej klasy mamy możliwość wstawiania wartości na daną pozycję, sortowania i odwracania kolejności elementów:

```
l = ["i", "m"]
l.insert(1, "c")
print(l)
l.reverse()
print(l)
l.sort()
print(l)
```

```
['i', 'c', 'm']
['m', 'c', 'i']
['c', 'i', 'm']
```

Zwróć uwagę że sortowanie i odwracanie modyfikuje istniejącą listę a nie tworzy kopii.

Zadanie 4.2.1

Napisz funkcję `zlicz` która dla podanej listy policzy powtórzenia jej elementów. Przykład użycia:

```
 > zlicz(["AX", "B", "AX"])
AX wystepuje 2 razy
B wystepuje 1 razy
```

Wskazówka: Użyj słownika, w którym element będzie stanowił klucz, a krotność jego wystąpień wartość. Możesz użyć metody ***`get()`*** *do pobierania wartości z słownika, jeżeli w nim jest lub wartości domyślnej w przeciwnym wypadku - szczegóły zobacz w dokumentacji*

4.3 Słowniki

Kolejnym użytecznym typem zmiennych w Pythonie są słowniki (zwane niekiedy *mapami* lub *tablicami asocjacyjnymi*). Podobnie jak listy służą do do przechowywania innych zmiennych. W odróżnieniu jednak od list w słownikach przechowywane są pary klucz - wartość, gdzie unikalny klucz służy do identyfikowania wartości. Zwróć uwagę na analogię z normalnymi słownikami klucz to słowo które wyszukujemy, a wartość to jego opis.

```
slownik = { "bd" : "xx", 5: True, "a" : 11 }
for klucz in slownik:
    print (klucz, "=>", slownik[klucz])
```

```
a => 11
bd => xx
5 => True
```

Zauważ że zarówno klucz, jak i wartość mogą być dowolnego typu oraz że słownik nie zachowuje kolejności dodawania elementów.

Możliwe jest także sprawdzanie istnienia jakiegoś elementu w słowniku, usuwanie, dodawanie i zmienianie elementów słowniku, itd (zwróć także uwagę na inną metodę wypisywania słownika - poprzednio iterowaliśmy po kluczach, teraz po liście par klucz-wartość):

```
if "bd" in slownik:
    print ("jest element o kluczu 'bd'")
    del slownik['bd']
slownik[15] = True
slownik["a"] = "yy"
for k,v in m.items():
    print (k, "=>", v)
```

```
jest element o kluczu 'bd'
a => yy
15 => True
```

4.3.1 Sortowanie słownika

Jak już wspomnieliśmy słownik nie zachowuje porządku elementów. Jeżeli chcemy uzyskać posortowaną listę kluczy, wartości lub par klucz-wartość z słownika możemy skorzystać z funkcji `sorted()`. W przypadku par wywołanie będzie wyglądać następująco:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
lista = sorted( mapa.items() )
print(lista)
```

```
[('5', 3), ('a', 101), ('bd', 20)]
```

Zwróć uwagę, iż użyliśmy tej samej metody `items()`, z której korzystaliśmy do iterowania po parach klucz-wartość (dla listy samych kluczy lub wartości należy użyć w tym miejscu innej metody klasy `dict`). Zapewne zauważyłeś że sortowanie zostało przeprowadzone w oparciu o klucze, co jednak jeżeli chcielibyśmy posortować taką listę w oparciu o wartości? W takim przypadku możemy skorzystać z opcjonalnego argumentu funkcji `sorted()` o nazwie `key`, który przyjmuje funkcję mającą za zadanie na podstawie otrzymanego elementu listy (w tym wypadku pary klucz - wartość) zwrócić klucz sortowania:

```
mapa = {'5': 3, 'bd': 20, 'a': 101}
def k(x):
    return x[1]
lista = sorted( mapa.items(), key=k )
print(lista)
```

```
[('5', 3), ('bd', 20), ('a', 101)]
```

4.4 Funkcje jako argumenty funkcji 😜

W powyższym przykładzie jednym za argumentów funkcji `sorted()` jest inna funkcja. Zauważ, że funkcja może być takim samym argumentem innej funkcji jak dowolna inna zmienna, może być też wynikiem zwracanym przez funkcję oraz może być przechowywana w zmiennej.

```
def dzialanie(operacja):
    if operacja == "dodaj":
        def f(a, b):
            return a+b
        return f
    elif operacja == "mnóż":
        def f(a, b):
            return a*b
        return f
def dwa(funkcja, argument):
    return funkcja(2, argument)

d = dzialanie("dodaj")
a = dwa(d, 11)
b = dzialanie("mnóż")(3,4)
print(a, b, d(3,4))
```

```
13 12 7
```

Zauważ że:

- wynikiem funkcji `dzialanie()` jest funkcja wykonująca wskazane działanie,
- funkcja `dwa()` jako argumenty przyjmuje funkcję realizującą działanie dwuargumentowe i jeden argument przekazywany do niej,
- zmienna d wskazuje na funkcję zwróconą przez funkcję `dzialanie()` i może być używana jako funkcja.

Zadanie 4.4.1

Napisz funkcję która przyjmuje dwa argumenty: listę oraz funkcję. Funkcja ma za zadanie wykonać przekazaną do niej funkcję na każdym elemencie listy. Przykład użycia:

```
>>> wykonaj([1,2,3], print)
1
2
3
```

Zadanie 4.4.2

Zastanów się czy konstrukcję if/elif w funkcji `dzialanie()` z rozdziału 4.4 można by zastąpić słownikiem, jak to ewentualnie zrobić i jakie mogłoby mieć to zalety bądź wady?

4.5 Zmienna, obiekt i referencja

W Pythonie każda zmienna jest nazwą wskazującą na jakiś obiekt w pamięci. Podobnie każdy element listy czy słownika wskazuje na jakiś obiekt[6]. Na jeden obiekt może wskazywać wiele zmiennych i/lub elementów innych obiektów (takich jak listy czy słowniki). Jeżeli zmienna nie ma na co wskazywać (np. został do niej przypisany wynik funkcji, która nie zwraca wartości) wskazuje na obiekt **`None`** (typu `NoneType`). Zatem na wszystkie zmienne pythonowe możemy patrzeć jak na referencje do obiektów istniejących gdzieś w pamięci.

Do uzyskania identyfikatora obiektu związanego z daną nazwą, lub elementem innego obiektu służy funkcja `id` (w przypadku standardowej implementacji Pythona jest to po prostu adres w pamięci).

4.5.1 Usuwanie i czas życia zmiennych

Instrukcja **`del`**, której używaliśmy już do usuwania elementów z listy lub słownika może być wykorzystana także do usuwania innych zmiennych. Należy jednak pamiętać iż w Pythonie usunięcie zmiennej nie wiąże się z natychmiastowym zwolnieniem zajmowanej przez nią pamięci z kilku powodów:

- na pojedynczy obiekt może wskazywać kilka zmiennych

6. Zasadniczo wszystkie definiowane przez nas zmienne czy funkcje są elementem słownika związanego z danym kontekstem. Do słowników tych można uzyskać dostęp poprzez funkcje `globals()` (słownik zawierający elementy zdeklarowane w kontekście globalnym) i `locals()` (słownik zawierający elementy zadeklarowane w kontekście lokalnym).

- to Python decyduje o tym kiedy zwalniać / ponownie użyć pamięć pozostałą po obiektach na które nie wskazuje już żadna nazwa

4.5.2 Kopiowanie obiektów

Python w momencie przypisania wartości jednej zmiennej do innej nie tworzy kopii obiektu na który wskazuje zmienna, zamiast tego przypisuje referencję do istniejącego obiektu. Jest to szczególnie zauważalne w obiektach, które mogą być wewnętrznie modyfikowalne (takich jak listy czy słowniki)[7]:

```
a = [1, 2, 3]
b = a
print(a, b, "\n", hex(id(a)), hex(id(b)))
a[1] = 0
print(a, b, "\n", hex(id(a)), hex(id(b)))
del a
print(b,    "\n", hex(id(b)))
```

```
[1, 2, 3] [1, 2, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3] [1, 0, 3]
 0x7f50d76b2bc8 0x7f50d76b2bc8
[1, 0, 3]
 0x7f50d76b2bc8
```

Jak widać `a` i `b` posiadają taki sam identyfikator obiektu zwracany przez funkcję `id`, modyfikacja `a[1]` wpłynęła na zawartość `b`, natomiast usunięcie `a` nie ma wpływu na `b` (usunęliśmy tylko jedną z dwóch referencji na wspólny obiekt). Jeżeli chcemy uzyskać kopię listy lub słownika musimy skorzystać z metody `copy()` odpowiedniego obiektu:

```
a = [1, 2, 3]
b = a.copy()
b[1] = "X"
print(a, b, "\n", hex(id(a)), hex(id(b)))
```

```
[1, 2, 3] [1, 'X', 3]
 0x7f50d76b2bc8 0x7f50d57a7088
```

Zauważ że tak utworzone `b` ma inny identyfikator obiektu niż `a`. Należy mieć także na uwadze że nawet argumenty funkcji przekazywane są jako referencje na obiekty a nie kopie obiektów, natomiast dopiero operacja przypisania nowej wartości do zmiennej związanej z argumentem powoduje że zaczyna ona wskazywać na nowo utworzony (w wyniku wyrażenia po prawej stronie znaku równości) obiekt.

7. Zauważ że jedyną możliwością modyfikacji liczby czy napisu jest przypisanie wartości wyrażenia do zmiennej, a dla list czy słowników możemy je modyfikować bez operacji przypisania całej listy czy słownika do nowej czy tej samej zmiennej. Jest to podział na typy "immutable" i "mutable" - te pierwsze nie są wewnętrznie modyfikowalne (każda modyfikacja odbywa się przez przypisanie obiektu do zmiennej, w wyniku którego pod zmienną może zostać podpięty nowy obiekt).

4.5.3 Lambda

```
tablicaA = [None, None, None, None]
tablicaB = [None, None, None, None]

for x in [0,1,2,3]:
        def tmpA(a):
                return x+a
        def tmpB(a, x=x):
                return x+a
        tablicaA[x]=tmpA
        tablicaB[x]=tmpB

print (tablicaA[1](3), tablicaB[1](3))
x = 0
print (tablicaA[1](3), tablicaB[1](3))
```

```
6 4
3 4
```

Python pozwala na definiowanie funkcji wewnątrz funkcji oraz łatwe przechowywanie funkcji w zmiennych, także kolekcjach takich jak listy i słowniki. Warto zwrócić uwagę na sposób obsługi zmiennych globalnych, czy też zewnętrznych w takich przypadkach. W powyższym przykładzie widzimy, że funkcje tak zdefiniowane nie korzystają z wartości zmiennej x z miejsca definicji, tylko z miejsca wywołania. Jeżeli potrzebujemy aby funkcja używała wartości z chwili definicji to możemy użyć x jako wartości domyślnej któregoś z argumentów tej funkcji.

Innym sposobem lokalnego definiowania funkcji, często dostępnym także w językach nie pozwalających na tak swobodne definiowanie zwykłych funkcji, jest tak zwana lambda. W tym przypadku definicja jest jeszcze bardziej lokalna, funkcja nawet nie posiada swojej nazwy. A taka lambda może być bezpośrednio przypisana do jakiejś zmiennej lub przekazana w argumencie. Warto zauważyć też że sposób traktowania zmiennych zewnętrznych jest analogiczny.

```
tablicaA = [None, None, None, None]
tablicaB = [None, None, None, None]

for x in [0, 1, 2, 3]:
        tablicaA[x] = lambda a,x=x: x+a
        tablicaB[x] = lambda a: x+a

print (tablicaA[1](3), tablicaB[1](3))
x = 0
print (tablicaA[1](3), tablicaB[1](3))
```

```
6 4
3 4
```

4.5.4 Dla jeszcze bardziej dociekliwych

Osobom jeszcze bardziej dociekliwym w temacie wnętrzności Pythona możemy polecić lekturę artykułu omawiającego te zagadnienia http://www.rwdev.eu/articles/objectthinking oraz samodzielne eksperymenty.

4.6 Klasy i struktury

Inną metodą grupowania zmiennych i funkcji jest definiowanie własnych klas:

```
class NazwaKlasy:
  # pola składowe
```

```
  a, d = 0, "ala ma kota"
  # metody składowe
  def wypisz(self):
    print(self.a + self.b)
  # metody statyczna
  @staticmethod
  def info():
    print("INFO")
  # konstruktor (z jednym argumentem)
  def __init__(self, x = 1):
    print("konstruktor", self.a , self.d)
    # i kolejny sposób na utworzenie pola składowego klasy
    self.b = 13 * x
```

Warto zauważyć jawny argument metod składowych klasy w postaci obiektu tej klasy. W innych językach programowania ten argument także występuje, ale często jest ukryty przed programistą - nie podajemy do ani w definicji metody, ani przy odwołaniach do pól klasy w metodzie (np. w C++).

Możliwe jest także dziedziczenie po jednej lub kilku klasach bazowych, w tym celu definicje klasy rozpoczynamy:

```
class NazwaKlasy(Bazowa1, Bazowa2):
```

Tworzenie obiektu klasy i używanie go:

```
k = NazwaKlasy()
k.a = 67
k.wypisz()
```

```
80
```

Obiekty można rozszerzać o nowe składowe i funkcje:

```
k.c = k.a + 10
print(k.c)
```

```
77
```

W ten sposób można też tworzyć całe struktury:

```
class Pusta():
  pass
x = Pusta()
x.a = 3
x.b = 4
```

Od strony implementacyjnej są one trzymane w słowniku związanym z danym obiektem o nazwie `__dict__`.
Spróbuj wypisać zawartość `x.__dict__` oraz `k.__dict__`.

Do metod klasy możemy odwoływać się także z podaniem nazwy klasy a nie obiektu, w takim wypadku jeżeli nie są to metody statyczne należy przekazać jako argument obiekt danej klasy lub go udający[8]:

```
NazwaKlasy.info()
NazwaKlasy.wypisz(k)
NazwaKlasy.wypisz(x)
```

```
INFO
80
7
```

Obiekty klas są obiektami modyfikowalnymi, zatem jak wiemy zwykłe przypisanie tworzy tylko inną referencję na ten sam obiekt. Celem utworzenia kopii naszego obiektu możemy zaimplementować własną metodę copy lub skorzystać z funckji copy dostarczanej przez moduł copy.

8. Wystarczy żeby taki obiekt miał metody i składowe używane przez dana metodę, nie musi to być obiekt tej klasy.

4.7 Iteratory i generatory 🤓

Iterator jest obiektem pozwalającym na dostęp do kolejnych elementów jakiejś kolekcji (np. listy). Są one przydatne np. gdy chcemy uzyskiwać kolejne elementy kolekcji nie iterując po niej w ramach pętli **for**. Jego użycie wygląda następująco:

```
l = [6, 7, 8, 9]
i = iter(l)  # zmienna i jest tutaj iteratorem
print( next(i) )
print( next(i) )
```

Niekiedy zamiast tworzenia listy lepsze może być uzyskiwanie jej kolejnych elementów "na żywo". Funkcjonalność taką w pythonie zapewniają generatory. Są to funkcje które zwracają kolejne elementy danej kolekcji używając słowa kluczowego **yield**, zamiast **return**. Pamiętają one też swój stan wewnętrzny pomiędzy wywołaniami w ramach poszczególnych iteracji.

Generatory możemy używać np. do iterowania po nich w pętli **for**, możemy tez używać iteratorów do pobierania kolejnych wartości z generatora:

```
def f(l):
    a, b = 0, 1
    for i in range(l):
        yield a
        a, b = b, a + b

ii = iter( f(8) )
for i in f(16):
    print("i =", i)
    if i > 6:
        print("ii =", next(ii))
```

Można także tworzyć generatory nieskończone:

```
def ff():
    a, b = 0, 1
    while True:
      yield a
      a, b = b, a + b
```

4.8 Obsługa błędów

Wcześniej spotkaliśmy się już z komunikatem błędu. Błędy mogą wynikać z błędów składniowych w programie ale również nie przewidzianych zdarzeń w trakcie jego pracy. Warto mieć na uwadze iż prawie wszystkie błędy w Pythonie mają postać wyjątków które mogą zostać obsłużone blokiem **try**/**except**.

```
try:
  a = 5 / 0
except ZeroDivisionError:
  print("dzielenie przez zero")
except:
  print("inny błąd")
```

Przy obsłudze błędów może przydać się instrukcja pusta **pass**, która w tym przypadku pozwala na zignorowanie obsługi danego błędu.

```
try:
  slownik["a"] += 1
except:
  pass
```

Powyższy kod zwiększy wartość związaną z kluczem "a" w słowniku slownik, jednak gdy napotka błąd (np. słownik nie zawiera klucza "a") zignoruje go.

Możemy także generować wyjątki z naszego kodu, służy do tego instrukcja **raise**, której należy przekazać obiektem dziedziczącym po **BaseException** np:

```
raise BaseException("jakiś błąd")
```

4.9 Pliki

Do tej pory wszystkie dane, z których korzystały nasze programy, wprowadzaliśmy bezpośrednio do kodu programu. W realnych zastosowaniach bardzo często użyteczniejsze jest korzystanie z danych zapisanych w osobnych plikach.

4.9.1 Zapisywanie tekstu do pliku

Zapis do pliku tekstowego możemy zrealizować w sposób następujący:

```
plik = open('dane.txt', 'wt', encoding='utf8')
plik.write("teskt1\n")
plik.write("teskt2\nteskt3")
plik.close()
```

Jak to działa?

- Polecenie z pierwszej linijki otwiera plik dane.txt i zapewnia dostęp do niego poprzez zmienną plik. Opcja 'w' oznacza, że plik jest otwarty „do zapisu” (od angielskiego *write*). Opcja 't' oznacza, że plik traktowany jako plik tekstowy[9]. Argument encoding pozwala na określenie kodowania użytego do zapisu pliku tekstowego, jest on opcjonalny i gdy nie zostanie podany kodowanie pliku zależne jest od ustawień systemowych.
- Druga i trzecia komenda zapisuje podany jako argument tekst do pliku dane.txt (zwróć uwagę na wstawianie nowej linii przy pomocy '\n')
- Ostatnie polecenie zamyka dostęp do pliku dane.txt.

Po uruchomieniu powyższego kodu powinien zostać utworzony plik „dane.txt”, zawierający 3 linie tekstu. Jeżeli plik taki wcześniej istniał zostanie on nadpisany.

4.9.2 Wczytywanie tekstu z pliku

```
plik = open('dane.txt', 'rt', encoding='utf8')
for linia in plik:
  print(linia, end="")
plik.close()
```

9. Tekst możemy zapisywać także do plików otwieranych jako binarne, w takim wypadku argument funkcji write musi mieć typ bytes a nie str, czyli być już jawnie zakodowanym w jakimś standardzie.

Zauważ, że została używa opcja `'r'` do otwarcia pliki co oznacza otwarcie do odczytu (od angielskiego *read*). Jeżeli chcemy wczytać cały plik do zmiennej napisowej możemy, zamiast pętli czytającej kolejne linie, użyć metody `read()`:

```
plik = open('dane.txt', 'rt', encoding='utf8')
napis = plik.read()
plik.close()
```

Po otwartym pliku możemy się przemieszczać metodą `seek`, na przykład `plik.seek(0)` przesunie punkt odczytu na początek pliku i umożliwi jego ponowne przeczytanie.

4.9.3 Czekanie na dane

Niekiedy nasz program musi poczekać na jakieś dane (np. wprowadzane z standardowego wejścia przez użytkownika). Typowo funkcje odczytu (takie jak `sys.stdin.read()`, `sys.stdin.readline()`, `input()`) czekają na koniec wczytywanych danych lub na koniec linii. Komplikacja pojawia się kiedy chcielibyśmy aby nasz program miał ograniczenie czasowe takiego oczekiwania lub czekał na pojawienie się danych w jednym z kilku źródeł. W takich przypadkach przydatna jest funkcja systemowa `select()`, którą w Pythonie znajdziemy w module *select*.

```
import sys, os, select

rdfd, _, _ = select.select([sys.stdin], [], [], 3.0)

if not rdfd:
        print("czas minął")

for fd in rdfd:
        print("czytam z:", fd)
        a = os.read(fd.fileno(), 1024)
        print("wczytałem:", a)
```

Funkcja `select()` przyjmuje 3 listy „deskryptorów plików” (czyli tego co zwraca np. funkcja `open()`) oraz ilość sekund, którą ma czekać na początek danych. Pierwsza lista związana jest z plikami z których chcemy czytać, druga pisać, a trzecia z plikami na których czekamy na wyjątkowe warunki. Funkcja ta zwraca również 3 takie listy, ale zawierające jedynie deskryptory plików na których pożądana operacja jest możliwa (np. są dane do wczytania, można zapisać dane).

Funkcja `select()` kończy działanie gdy pojawią się jakiekolwiek dane (nie czeka na koniec danych – EOF). Zauważ, że do odczytu zastosowana została funkcja `os.read()` a nie metoda `fd.read()`, wynika to z faktu, iż `fd.read()` czeka na EOF lub podaną ilość bajtów, a `os.read()` wczytuje to co jest dostępne i ogranicza jedynie maksymalną ilość wczytywanych danych (resztę możemy doczytać kolejnym wywołaniem).

Zadanie 4.9.1

Napisz funkcję, który wczytuje dane z standardowego wejścia. Funkcja powinna przyjmować jeden argument określający maksymalny czas oczekiwania na kolejną porcję danych. Każde pojawienie się danych wejściowych powinno resetować odliczanie timeoutu podanego w argumencie. Po skutecznym upływie tego timeoutu funkcja powinna zwrócić wszystkie wczytane dane.
Wskazówka: zmodyfikuj przykład użycia funkcji select() podany w skrypcie.

Zadanie 4.9.2

Napisz funkcję `zapisz` która przyjmuje dwa argumenty: słownik oraz nazwę pliku. Funkcja ma utworzyć plik o podanej nazwie i zapisać do niego otrzymany słownik, w taki sposób że każda linii odpowiada jednej parze klucz wartość, a separatorem pomiędzy kluczem a wartością jest znak tabulacji. Dla uproszczenia zakładamy że elementy słownika są napisami (zarówno klucze jak i wartości) i nie zawierają znaków nowej linii ani tabulacji.
Na przykład dla wywołania `zapisz({"a": "qwe", "d": "123"}, "xx")` funkcja powinna utworzyć plik z zawartością:

```
a	qwe
d	123
```

4.10 Kod binarny

Jak wiemy liczby możemy zapisywać w różnych systemach liczbowych i jednym z nich jest system dwójkowy, nazywany też binarnym. Taka reprezentacja liczb jest podstawą działania elektroniki cyfrowej w tym współczesnych komputerów.

Napis przedstawiający liczbę w reprezentacji dwójkowej w Pythonie można z pomocą funkcji `bin`. Funkcja ta niestety nie pozwala wymusić długości wypisywanej liczby (co jest bardzo przydatne jeżeli chcemy operować na poszczególnych bitach) a dodatkowo liczby ujemne wypisuje ze znakiem minus i reprezentacją liczby dodatniej (czyli zasadniczo w kodzie znak moduł) a nie rzeczywiście stosowanym do zapisu takich liczb na zdecydowanej większości architektur kodzie uzupełnień do dwóch. Dlatego na potrzeby przykładów w tym rozdziale będziemy używać własnej funkcji zwracającej binarną reprezentację liczb 8bitowych (czyli 1 bajta):

```
def bin8(x):
        return "0b{0:08b}".format(x & 0xff)
```

Liczby dodatnie w systemie binarnym zapisuje się praktycznie zawsze w postaci NKB. Zapis taki jest analogiczny do zapisu dziesiętnego stosowanego na co dzień, z tym że kolejne cyfry liczby mają wagę 2^n a nie 10^n (gdzie n jest numerem cyfry, zaczynającym się od zera dla skrajnie prawej).

$$a_n a_{n-1} ... a_1 a_0 \leftrightarrow a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Liczby ujemne mogą być zapisywane na różne sposoby. Wspomniany kod moduł-znak polega na zapisie modułu liczby w postaci NKB oraz umieszczenia flagi znaku w najstarszym bicie (0 – liczba dodatnia, 1 – ujemna). Najczęściej stosowany jest jednak kod uzupełnień do dwóch (określany jako U2) przypominający NKB tyle że najstarszy n-ty bit wchodzi z wagą $-(2^n)$ a nie 2^n:

$$a_n a_{n-1} ... a_1 a_0 \leftrightarrow -a_n \cdot 2^n + a_{n-1} \cdot 2^{n-1} + ... + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

```
print(bin8(3),  bin(3))
print(bin8(-3), bin(-3))
```

```
0b00000011 0b11
0b11111101 -0b11
```

Możemy sprawdzić czy bin8 rzeczywiście wypisało reprezentację -3 w kodzie U2 wykonując proste obliczenie:

```
-2**7 + 2**6 + 2**5 + 2**4 + 2**3 + 2**2 + 0*(2**1) + 2**0
```

```
-3
```

4.10.1 Operacje bitowe

Python, jak wiele innych języków, pozwala wykonywać operacje boolowskie nie tylko na wartościach reprezentujących pradwę i fałsz, ale także na odpowiadajacych sobie bitach dwóch liczb. Operację bitowego

AND zapisujemy z pomocą &, OR z pomocą |, XOR z pomocą ^, a NOT z pomocą ~:

```
print(bin8( 0b11001010 & 0b10101110 ))
print(bin8( 0b11001010 | 0b10101110 ))
print(bin8( 0b11001010 ^ 0b10101110 ))
print(bin8( ~0b11001010 ))
```

```
0b10001010
0b11101110
0b01100100
0b00110101
```

Jak widzimy w pokazanym przykładzie operacje te są wykonywane na każdym z bitów liczby niezależnie czyli n-ty bit wyniku bitowego AND to n-ty bit pierwszej liczby AND n-ty bit drugiej liczby, itd.

```
print(bin8( 0b11001010 << 3 ))
print(bin8( 0b11001010 >> 3 ))
```

```
0b01010000
0b00011001
```

Dostępne są także operacje przesunięcia bitów w ramach liczby w lewo lub prawo (brakujące bity uzupełniane są zerami, a bity wystające poza długość liczby binarnej są obcinane[10]). Operacje te odpowiadają mnożeniu i dzieleniu całkowitemu przez 2^x, gdzie x to ilość bitów do przesuniecia podawana po prawej stronie operatora przesuniecia w postaci << lub >>.

Operacje takie są przydatne do sprawdzania bądź ustawiania wartości poszczególnych bitów. Są to operacje dość niskopoziomowe i nie często stosowane w Pythonie, ale wiedza o nich przyda nam się w niedalekiej przyszłości.

5 Podstawy programowania równoległego

5.1 procesy i fork()

Aby w systemie mógł działać więcej niż 1 proces konieczna jest możliwość utworzenia nowego procesu (potomka) z poziomu procesu aktualnie działającego (rodzica). Możliwe są dwa podejścia:

- utworzenie "czystego" procesu uruchamiającego podany kod programu z podanymi argumentami (spawn)
- utworzenie kopii aktualnego procesu, która zacznie wykonywać się niezależnie od momentu rozgałęzienia (fork)

W przypadku zastosowania fork proces potomny otrzymuje kopię pamięci rodzica (ma dostęp do wszystkich jego zmiennych oraz zasobów uzyskanych przed fork(); dalsze operacje na zmiennych są niezależne). Po utworzeniu kopi procesu można (ale nie trzeba) zastąpić wykonywany w nim program innym poprzez funkcje z rodziny exec. Cechy te powodują że mechanizm fork jest bardziej elastyczny od spawn.

```
import os

print("pid to:", os.getpid())

pid = os.fork()
if pid == 0:
        print("potomek: mój pid to", os.getpid())
else:
        print("rodzic: pid potomka to", pid)
```

```
pid to: 8763
rodzic: pid potomka to: 8764
potomek: mój pid to 8764
```

Przykład możemy trochę rozbudować używając funkcji sleep aby zaobserwować współistnienie tych dwóch procesów oraz funkcji signal do zakończenia procesu potomnego przez rodzica:

10. W przypadku Pythona liczby całkowite nie mają maksymalnej wielkości, a obcinanie przy przesuwaniu w lewo realizuje nasza funkcja wypisująca bin8.

```python
import os, time, signal

print("pid to:", os.getpid())

pid = os.fork()
if pid == 0:
        print("potomek: mój pid to", os.getpid())
        time.sleep(4)
        print("potomek 1")
        time.sleep(7)
        print("potomek 2")
else:
        print("rodzic: pid potomka to", pid)
        time.sleep(5)
        print("rodzic 1")
        time.sleep(4)
        print("zabijam potomka")
        os.kill(pid, signal.SIGTERM)
        time.sleep(5)
        print("rodzic 2")
```

```
pid to: 5295
rodzic: pid potomka to 5301
potomek: mój pid to 5301
potomek 1
rodzic 1
zabijam potomka
rodzic 2
```

Zadanie 5.1.1

Napisz program który utworzy 1 potomka, rodzić powinien wypisać PID potomka i swój. Natomiast potomek powinien utworzyć kolejny proces w którym zostanie uruchomiona komenda `ps -f` w taki sposób aby potomek odebrał do zmiennej jej standardowe wyjście i wypisał je na ekran.

5.2 wywołanie zewnętrznej komendy

Najprostszym sposobem uruchomienia innej komendy z poziomu Pythona jest użycie funkcji `system()` z modułu `os`:

```python
import os

inStr = "Ala ma kota\nKot ma psa\n..."

os.system('echo -en "' + inStr + '" | grep -v A')
```

Jak widać przekazujemy do niej napis takiej samej postaci jak wyglądałby komenda uruchamiana w terminalu. Mechanizm ten nie daje jednak zbyt dużej kontroli nad uruchamianiem tego polecenia (nie pozwala na proste odebranie jego standardowego wyjścia, przekazanie wejścia również wymaga dodatkowego zabiegu w postaci dodania komendy echo, itd.). Bardziej elastycznym rozwiązaniem jest pythonowy moduł *subprocess*:

```python
import subprocess

inStr = "Ala ma kota\nKot ma psa\n..."

# uruchamiamy subprocess z grep'em
res = subprocess.run(["grep", "-v", "A"], input=inStr.encode(),
 ↪  stdout=subprocess.PIPE)
```

```
print("Kod powrotu to: " + str(res.returncode))
print("Standardowe wyjście z komendy to: " + res.stdout.decode())
# warto zwrócić uwagę na kodowanie i dekodowanie napisów
# (przekazywanych/odbieranych przez stdin/stdout) do / z utf-8

# jeżeli chcemy korzystać np. z znaków uogólniających powłoki lub podać
# komendę jako pojedynczy napis (a nie listę argumentów) to można użyć
# opcji shell=True:
subprocess.run(["ls -ld /etc/pa*"], shell=True)
# jeżeli potrzebujemy tylko rozbicia napisu na listę argumentów można
# użyć shlex.split()

# run() pozwala także (obok subprocess.PIPE) na przekazywanie
# istniejących deskryptorów (lub subprocess.DEVNULL, co ignoruje wyjście)
# w ramach stdin, stdout, stderr

# moduł subprocess oferuje także funkcję Popen() dającą większą kontrolę
# nad uruchamianiem komendy
```

5.3 komunikacja międzyprocesowa

W systemie wieloprocesowym konieczne jest zapewnienie mechanizmów komunikacji pomiędzy procesami, zwłaszcza jeżeli grupa procesów ma realizować wspólne zadanie.

Jednym z takich mechanizmów (można powiedzieć że nawet podstawowym) jest poznane już wcześniej łącze nie nazwane (pipe, uzyskiwane np. w bashowej linii poleceń przy pomocy `|`) pozwalające na przekazywanie strumienia danych od jednego do kolejnego procesu. Podobnie działa łącze nazwane z tym że nie jest uzyskiwane w wyniku funkcji `pipe()` a otwarcia specjalnego pliku (utworzonego `mkfifo()`) przez dwa procesy (jeden do czytania drugi do pisania).

Innymi mechanizmami komunikacji międzyprocesowej są m.in:

- sygnały
- kolejki komunikatów
- pamięć współdzielona

Stosowanie pamięci współdzielonej wymaga często też stosowania mechanizmów ochrony dostępu do niej (wejścia do „krytycznych sekcji” kodu). Koncepcja takiej ochrony wygląda następująco:

```
if !blokada:
        blokada = True
        # działania na pamięci wspólnej
        blokada = False
```

Jednak nie może być zrealizowana w tak prosty sposób, gdzyż przełączenie procesów może nastąpić pomiędzy sprawdzeniem warunku na zmiennej `blokada` a zmianą jej wartości (lub mogą one działać idealnie równolegle i w tym samym momencie sprawdzać wartość zmiennej `blokada`). Dlatego do ochrony sekcji krytycznych stosuje się mechanizmy systemowe takie jak semafory i lock'i.

5.4 wątki

Oprócz możliwości pełnego rozgałęzienia procesu (utworzenia potomka), możliwe jest także tworzenie wątków (zwanych też lekkimi procesami) w ramach bieżącego procesu. Wątek (w odróżnieniu od procesu potomnego) korzysta z tej samej pamięci (przestrzeni adresowej) co oryginalny proces i wszystkie inne jego wątki (czyli *out of the box* mają pamięć współdzieloną). Jednak każdy wątek posiada niezależny stos (umieszczany w innym fragmencie współdzielonej pamięci), który jest używany m.in. do przechowywania

zmiennych lokalnych (w tym argumentów funkcji), czyli dopóki ograniczamy się do zmiennych lokalnych nie ma potrzeby stosowania ochrony sekcji krytycznych ze względu na dostęp do pamięci.

5.5 „Python-way"

Zaprezentowane powyżej podejście korzysta w dużej mierze z funkcji analogicznych do funkcji systemowych biblioteki standardowej C zgromadzonych w module *os*. Python oferuje obok wspomnianego modułu *subprocess* także inne własne mechanizmy związane z tworzeniem wielu procesów poprzez moduł *multiprocessing* oraz oferuje wsparcie dla wątków w module *threading*[11]. Jednak, jako że w ramach tego kursu nie będziemy zajmować się programowaniem równoległym jako takim, to modułów tych nie omówimy w tym skrypcie ani na zajęciach. Zainteresowanym polecam zapoznanie się z http://vip.opcode.eu.org/#Procesy_i_wątki.

6 Biblioteki

Ideą korzystania z funkcji w trakcie tworzenia programu jest zapewnienie jego większej czytelności oraz unikanie powtarzania kodu robiącego to samo w wielu miejscach programu – kod umieszczamy w funkcji którą tylko wywołujemy z odpowiednimi argumentami i odbieramy wynik działania (np. poprzez zwracaną wartość). Rozwinięciem tej ideii są biblioteki stanowiące zbiory funkcji oraz struktur danych (własnych typów zmiennych) służących do realizacji określonych zadań.

Reguły DRY i KISS

„Don't Repeat Yourself" (*nie powtarzaj się*) jest jedną z dwóch głównych reguł programistycznych (ale ma także pewne zastosowania w innych dziedzinach techniki). Zaleca ona unikanie potarzania tych samych czynności, czy też tworzenia takich samych, a nawet analogicznych, podobnych fragmentów kodu.
Narzędziami ułatwiającymi realizację tego celu są m.in.:

- systemy i skrypty służące automatyzacji różnego rodzaju czynności (takich jak np. kompilacja, instalacja, aktualizacja, monitoring działania) – zarówno systemy takie jak make, cmake, doxygen ale również wszystkie drobne skrypty (np. shellowe czy pythonowe) tworzone w tym celu w codziennej pracy informatyka
- elementy składniowe (m.in. takie jak pętle i funkcje) oraz mechanizmy (np. polimorfizm) dostępne w językach programowania pozwalające na eliminację powtórzeń kodu
- biblioteki, moduły, itp pozwalające na współdzielenie tych samych rozwiązań, tego samego kodu, pomiędzy różnymi projektami
- elementy biblioteki systemowej pozwalające na wywoływanie innych programów (np. exec) i komunikację z nimi (np. poprzez strumienie wejścia/wyjścia)

Unikanie powtórzeń takiego samego lub (co często nawet gorsze) tylko nieznacznie zmienionego kodu jest też szczególnie istotne ze względu na łatwość utrzymania kodu – np. jakąś poprawkę wprowadza się tylko w odpowiednio sparametryzowanej funkcji, a nie kilkunastu podobnych (ale nie identycznych, ze względu na brak parametryzacji) fragmentach kodu.
W zastosowaniach nie programistycznych przejawia się często wydzielaniem modułów i dążeniem do ich powtarzalności, redukcji ilości ich typów (np. dzięki parametryzacji, czy konfigurowalności).

Drugą, nawet chyba ważniejszą, z tych dwóch reguł jest **„Keep It Simple, Stupid"** (niekiedy *Keep It Small and Simple*), którą można streścić jako *proste jest lepsze*. Reguła KISS jest bardziej ogólna (można nawet powiedzieć że wynika z niej reguła DRY), posiada dużo szersze pole zastosowań (także nie technicznych) i może być uważana za implementację *Brzytwy Ockhama* w inżynierii. Zaleca ona m.in.:

- tworzenie przejrzystych, czytelnych i prostych rozwiązań (zarówno pod względem samego projektu, koncepcji, jak też ich implementacji, wykonania)

11. Należy mieć na uwadze iż pythonowe wątki są niepełnowartościowe - ze względu na konstrukcję interpretera CPython, jedynie jeden wątek w danej chwili może być aktywny - wykorzystywać CPU, pozostałe mogą jedynie czekać.

- wybór rozwiązania prostszego spośród (równie) skutecznych rozwiązań jakiegoś problemu
- myślenie o łatwości późniejszego utrzymania i serwisu tworzonego rozwiązania (czy to kodu programu, czy urządzenia elektronicznego, a nawet budynku)

W duchu prostoty nakazywanej regułą KISS należy także starać się trzymać powszechnie stosowanych standardów (gdy tylko jest to możliwe i nie powoduje zbyt wielkiej komplikacji naszego projektu), zamiast każdorazowo tworzyć nowe, własne standardy, protokoły czy interfejsy.

Do tej pory korzystaliśmy z elementów standardowej biblioteki dostarczanej z Pythonem. W rozdziale tym zaprezentujemy kilka różnych przykładowych bibliotek (w tym wchodzących w skład biblioteki standardowej Pythona), jednak żadnej z nich nie będziemy tutaj szczegółowo omawiać, gdyż nie miałoby to większego sensu. Istnieje ogromna liczba bibliotek dedykowanych różnym celom (obsługa formatów plików, standardów komunikacyjnych, tworzenie grafiki, ...) i nie ma sensu uczyć się ich bez realnej potrzeby zastosowania – programowanie w dużej mierze polega na wyszukiwaniu właściwych bibliotek, zapoznawaniu się z ich dokumentacją i wykorzystywaniu ich w własnych programach. W przypadku Pythona biblioteki najczęściej mają postać modułów pythonowych, które włączamy poprzez deklarację `include`.

Poniższe przykłady służą głównie zaprezentowaniu potencjału możliwego do uzyskania dzięki dostępnym bibliotekom, tego że są one dużym ułatwieniem dla programisty oraz pokazaniu kilku standardów związanych z zapisem danych.

6.1 XML

Extensible Markup Language (XML) jest tekstowym formatem wymiany danych. W odróżnieniu od formatu klasycznego formatu utożsamiającego linię z rekordem złożonym z pól oddzielanych wskazanym separatorem może on w łatwy sposób opisywać bardziej złożoną (drzewiastą a nie tabelkową) postać danych. Dokument XML składa się z zagnieżdżonych w sobie znaczników, każdy z nich może posiadać atrybuty oraz wartość, którą jest tekst zawierający lub nie kolejne znaczniki. Kolejność występowania elementów w dokumencie jest znacząca. Każdy znacznik otwierający posiada odpowiadający mu znacznik zamykający (np. `<b>aa</b>`), znaczniki bez wartości mogą być samo-zamykające (np. `<g />`). Dokumenty HTML mogą być zgodne z wymogami formalnymi XML tym samym stanowiąc dokumenty XML.

Do obsługi XML w Pythonie można skorzystać np. z modułu *ElementTree* (ale nie jest on jedyną biblioteką której możemy użyć):

```
import xml.etree.ElementTree as xml

txt = """<a>
        <b>A<h>qwe ... rty</h></b> ABCD... &amp;&apos; HIJ...
        <c x="q" w="p p">EE FĄ</c> <g y="zz" />
        <c x="pp">123 <d rr="oo">456</d> 78 90.</c>
</a>"""

rootNode = xml.fromstring(txt)

print("nazwa głównego elementu to:", rootNode.tag)
print("jego potomkowie to:")
for subNode in rootNode:
        print(" ", subNode.tag, ":", xml.tostring(subNode, encoding="unicode"))

# możemy pobrać listę potomków o określonej nazwie
# albo od razu po nich iterować pętlą for subNode in rootNode.iter("c"):
cSubNodes = list( rootNode.iter("c") )
if cSubNodes:
        for subNode in cSubNodes:
                print('element "c" ma atrybuty': subNode.attrib
```

```
else
        print('nie ma elementów "c"')

# możemy też używać iteratorów bezpośrednio, np:
print("pierwszy węzeł c ma atrybuty:")
try:
        ci = rootNode.iter("c")
        print(next(ci).attrib)
except StopIteration:
        print(" [brak takiego węzła]")
```

ElementTree pozwala też na modyfikowanie XMLa poprzez zmianę/dodawanie/usuwanie atrybutów, czy też całych tagów.

Innym sposobem zapisu ustrukturyzowanych danych w postaci tekstowej jest JSON. Przypomina on trochę output funkcji print z podanym do niej słownikiem lub listą. Do jego obsługi w Pythonie służy moduł *json*:

```
import json, pprint

a='''{
        "info": "bbb",
        "ver": 31,
        "d": [
                {"a": 21, "b": {"x": 1, "y": 2}, "c": [9, 8, 7]},
                {"a": 17, "b": {"x": 6, "y": 7}, "c": [6, 5, 4]}
        ]
}'''

# interpretacja napisu jako zbioru danych w formacie json
d = json.loads(a)

# wypisanie zbioru danych
pprint.pprint(d) # pprint ładnie formatuje złożone zbiory danych

# jak widać jest to zagnieżdżona struktura list i słowników
# odpowiadająca 1 do 1 temu co było w napisie

# dostęp do poszczególnych elementów: "po pythonowemu"
print(d["d"][1]["b"])
d["d"][1]["b"]["x"] = "XXX"

# wygenerowanie json'a w oparciu o zmienną pythonową
c = json.dumps(d, ensure_ascii=False)
print(c)
```

6.2 SQL

Innym sposobem przechowywania danych niż w postaci plików tekstowych są systemy baz danych. Standardowym językiem używanym do komunikacji z systemami bazodanowymi jest SQL. Pomimo jego standaryzacji istnieją różnice w składni zapytań dla poszczególnych silników bazodanowych (takich jak: MariaDB, PostgreSQL, SQLite, ...).

Typowo komunikacja z bazą danych odbywa się za pośrednictwem biblioteki odpowiedzialnej za nawiązanie połączenia z serwerem i przekazywanie do niego zapytań SQL. Wymaga to działania osobnego procesu (często nawet na innej maszynie) obsługującego silnik bazodanowy, co jest pożądanym rozwiązaniem dla baz danych z których równocześnie może korzystać wielu klientów. Typowym przykładem może być komunikacja skryptów jakiegoś serwisu interetowego z bazą danych.

Jednak takie podejście nie jest wygodne w rozwiązaniach nie wymagających współdzielenia bazy danych. Do zastosowań takich można użyć biblioteki SQLite, która pozwala na łatwe stosowanie bazy SQLowej do wewnętrznych potrzeb aplikacji, bez konieczności uruchamiania osobnego systemu bazodanowego. SQLite można wykorzystywać także bezpośrednio z poziomu Pythona, dzięki modułowi *sqlite3*:

```
import sqlite3
import os.path

if os.path.isfile('example.db'):
        create = False
else:
        create = True

conn = sqlite3.connect('example.db')
c = conn.cursor()

if create:
        print("create new db")
        c.execute("CREATE TABLE users (uid INT PRIMARY KEY, name TEXT);")
        c.execute("CREATE TABLE posts (pid INT PRIMARY KEY, uid INT, text TEXT);")

        c.execute("INSERT INTO users VALUES (21, 'user A');")
        c.execute("INSERT INTO users VALUES (2671, 'user B');")

        c.execute("INSERT INTO posts VALUES (1, 21, 'abc ..');")
        c.execute("INSERT INTO posts VALUES (2, 21, 'qwe xyz');")
        c.execute("INSERT INTO posts VALUES (3, 2671, 'test');")

        conn.commit()

maxUid = 100
for r in c.execute("SELECT * FROM users WHERE uid < ?;", (maxUid,)):
        print(r)

for r in c.execute("SELECT u.name, p.text FROM users AS u JOIN posts AS p ON (u.uid
 ↪  = p.uid);"):
        print(r)
```

6.3 GUI

Przykłady użycia 3 różnych graficznych interfejsów użytkownika z poziomu Pythona można znaleźć na http://vip.opcode.eu.org/#Graficzny_interfejs_użytkownika. W odróżnieniu od poprzednich przykładów, te biblioteki nie wchodzą w skład pythonowskiej biblioteki standardowej i mogą wymagać doinstalowania odpowiednich pakietów oprogramowania.

7 Wykład wideo[12]

- *Python: wprowadzenie* – http://video.opcode.eu.org/02.01-python-intro.mkv
- *Python: funkcje* – http://video.opcode.eu.org/02.02-python-funkcje.mkv
- *Python: pętle i warunki* – http://video.opcode.eu.org/02.03-python-petle_warunki.mkv
- *Python: napisy* – http://video.opcode.eu.org/02.04-python-napisy.mkv
- *Python: wyrażenia regularne* – http://video.opcode.eu.org/02.05-python-wyrazenia_regularne.mkv
- *Python: argumenty linii poleceń* – http://video.opcode.eu.org/02.06-python-argv.mkv
- *Python: listy i słowniki* – http://video.opcode.eu.org/02.07-python-listy_i_slowniki.mkv
- *Python: operacje bitowe* – http://video.opcode.eu.org/03.01-python-bitowe.mkv
- *Python: typy i referencje* – http://video.opcode.eu.org/03.02-python-typy_i_referencje.mkv
- *Python: klasy, wyjątki i generatory* – http://video.opcode.eu.org/03.03-python-klasy_wyjątki_i_generatory.mkv
- *Python: pliki i czekanie na dane* – http://video.opcode.eu.org/03.04-python-pliki.mkv
- *Python: procesy potomne (fork i exec)* – http://video.opcode.eu.org/03.05-python-równoległe.mkv
- *Python: biblioteki (xml, json, sql, ...)* – http://video.opcode.eu.org/03.06-python-biblioteki.mkv

8 Literatura dodatkowa 🤓

- *The Python Tutorial* (https://docs.python.org/3/tutorial/) - oficjalny Tutorial Pythona.
- *Biblioteka Riklaunima: Podstawy Pythona* (http://www.python.rk.edu.pl/w/p/podstawy/).
- *A Byte of Python* (https://python.swaroopch.com/).
- *How to Think Like a Computer Scientist: Learning with Python 3* (http://openbookproject.net/thinkcs/python/english3e/).
- *Zanurkuj w Pythonie* (https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie).

12. Filmy posiadają napisy wgrane do kontenera multimedialnego jako osobny strumień – napisy mogą być włączone lub wyłączone w odtwarzaczu. W wielu filmach dużo dzieje się "na dole ekranu", dlatego polecamy odtwarzać filmy z napisami umieszczonymi poniżej filmu, np. przy pomocy polecenia: `vlc --video-filter='croppadd{paddbottom=120}' --sub-margin=-10 PLIK.mkv`

9 Rozwiązania zadań

Treści zadań zamieszczone zostały w odpowiednich rozdziałach skryptu.

Poniżej zamieszczone są przykładowe rozwiązania „głównych” zadań z tego skryptu wraz z komentarzami. Wiemy że zajrzenie do nich już przy pierwszej trudności jest kuszące, mimo to rekomendujemy przynajmniej podjąć uczciwą, co najmniej kilkunastominutową na każde z zadań, próbę rozwiązania tych zadania bez zaglądania do odpowiedzi.

Pamiętaj!: Samodzielne rozwiązanie problemu (wraz z wszystkimi trudnościami po drodze i popełnionymi błędami) jest dużo bardziej kształcące od nawet wielokrotnego przepisania gotowego rozwiązania, jednak nawet jednokrotne przepisanie rozwiązania jest bardziej kształcące od wielokrotnego przekopiowania go.

Rozwiązanie zadania 2.1.1

```
def suma(a, b):
    return a + b

a = suma(17, 15)
print(a, suma(13, 16), suma(a, 11))
```

Zwróć uwagę na użycie słowa kluczowego **return** do zwrócenia wartości z funkcji. W odróżnieniu od bezpośredniego wpisania wyniku na ekran z użyciem np. `print` pozwala to m.in. na przechowanie tego wyniku w zmiennej i użycie w dalszych obliczeniach, co zostało zademonstrowane w drugiej części przykładu.

Rozwiązanie zadania 2.3.1

```
sum = 0
for x in range(101):
    sum = sum + x**2
print(sum)
```

Zwróć uwagę na użycie zewnętrznej w stosunku co do pętli zmiennej `sum`, służącej do przechowywania wartości modyfikowanej w każdym obiegu pętli (wyniku sumy) – jest to typowy schemat rozwiązywania tego typu problemów programistycznych.

Rozwiązanie zadania 2.5.1

```
def znak(liczba):
    if liczba > 0:
        print(liczba, "jest dodatnia")
        return liczba
    elif liczba < 0:
        print(liczba, "jest ujemna")
        return -liczba
    else:
        print(liczba, "to zero")
        return 0
```

Zadanie można by rozwiązać także używając funkcji obliczającej wartość bezwzględną (`abs()`), jednak ze względu na konstrukcję zadania musimy sami ustalić znak liczby, natomiast użycie tej (i tak już posiadanej) informacji do obliczenia wartości bezwzględnej jest wydajniejsze niż kolejne sprawdzanie znaku wewnątrz funkcji `abs()`.

Rozwiązanie zadania 2.6.1

```
sum = 0
x = 1
```

```
while x <= 100:
    sum = sum + x**2
    x = x + 1
print(sum)
```

Zauważ że w tym wariancie zadania potrzebujemy dwóch zmiennych zewnętrznych w stosunku co do pętli – jednej do przechowywania obliczanej sumy, a drugiej do przechowywania numeru kroku (wartości którą sumujemy). Ta druga zmienna w rozwiązaniu z użyciem pętli `for` jest dostarczana przez samą konstrukcję tamtej pętli. W przypadku pętli `while` to my jako twórcy kodu musimy ją zainicjalizować i zwiększać w każdym kroku. Pozwala to jednak na stosowanie bardziej zaawansowanych mechanizmów modyfikowania tej zmiennej.

Rozwiązanie zadania 3.0.1

```
def wskap(lista):
    for slowo in lista:
        # w pythonie zamiast ponizszej petli można prościej ...
        # ale warto poznać (także) takie rozwiązanie
        for i in range(len(slowo)):
            print(slowo[-1 - i], end = '')
        print()
```

Prostszym rozwiązaniem (nie wymagającym jawnego pisania pętli w pętli) jest:

```
def wskap(lista):
    for slowo in lista:
        print(slowo[::-1])
```

które korzysta z odwrócenia napisu przy pomocy pobrania wszystkich jego elementów z krokiem -1 poprzez `slowo[::-1]`

Rozwiązanie zadania 3.1.1

```
def wyiksuj(napis):
    duzy_alfabet = 'AĄBCĆDEĘFGHIJKLŁMNŃOÓPRSŚTUWYZŹŻ'
    maly_alfabet = 'aąbcćdeęfghijklłmnńoópqrsśtuwyzźż'
    wynik = ''
    for c in napis:
        if c in duzy_alfabet:
            wynik += 'X'
        elif c in maly_alfabet:
            wynik += 'x'
        else:
            wynik += c
    return wynik
```

inne rozwiązanie:

```
def wyiksuj(napis):
    wynik = ''
    for c in napis:
        if c.isupper():
            wynik += 'X'
        elif c.islower():
            wynik += 'x'
        else:
            wynik += c
    return wynik
```

jeszcze inne rozwiązanie (w tej formie obsługuje tylko litery ASCII, ale aktualna wersja zadania to dopuszcza):

```
def wyiksuj(napis):
  import re
  napis = re.sub("[a-z]", "x", napis)
  return re.sub("[A-Z]", "X", napis)
```

Zwróć uwagę że:

- iterowanie po elementach napisu (znakach) z użyciem pętli `for`
- zastosowanie konstrukcji `a in b` do sprawdzenia czy element a (w tym wypadku znak) należy do kolekcji b (w tym wypadku napisu, ale mogła by to być także np. lista znaków)
- zastosowanie metod `isupper()` i `islower()` w drugim wariancie rozwiązania, podobne porównanie dla znaków ASCI można łatwo wykonać w oparciu o wartość numerycznego kodu tego znaku
- zwięzłość rozwiązania z użyciem wyrażeń regularnych

Rozwiązanie zadania 3.2.1

```
def parse(t):
    t = t.replace("XY", " ")
    return t.split(":")
```

Zwróć uwagę że:

- używamy metody `replace` na oryginalnym napisie celem zastąpienie XY spacją, należy zauważyć że metoda ta nie modyfikuje oryginalnego napisu tylko zwraca nowy (zmieniony) napis, dlatego zapisujemy jej wynik do zmiennej, celem dalszego przetwarzania
- używamy metody split z określeniem separatora w jej argumencie, zwraca ona listę powstałą z podziału napisu przy pomocy wskazanego separatora
- listę tą zwracamy z funkcji za pomocą return

Rozwiązanie zadania 3.4.1

```
import codecs
d = b'UHl0aG9uIGplc3QgZmFqbnkg8J+Yjg==\n'
d = codecs.decode(d, 'base64')
d = d.decode()
print(d)
```

Zakodowany tekst to: Python jest fajny 😎
Zwróć uwagę że:

- zdejmowanie kolejnych kodowań w kolejnych krokach procedury – w odwrotnej kolejności niż były nakładane
- funkcja `codecs.decode` wymaga jako danych wejściowych ciągu bajtowego, i taki ciąg zwraca
- metoda `decode` ciągu bajtowego zwraca napis powstały przez zdekodowanie tego ciągu z użyciem utf-8

Rozwiązanie zadania 3.5.1

```
import re
def spr(x):
  if re.search("^[^ ]*$", x):
    print(x, "jest słowem")
  else:
    print(x, "NIE jest słowem")
```

Zadanie polega przede wszystkim na wymyśleniu odpowiedniego wyrażenia regularnego. Ze względu że funkcja `search` dopasowuje dowolny fragment napisu (wymaga aby napis zawierał fragment opisany podanym wyrażeniem regularnym), to nasze wyrażenie musi rozpoczynać się od ^ konczyć się $, aby wyrażenie było dopasowywane do całości sprawdzanego napisu. Zastosowane wyrażenie wymaga aby napis nie zawierał spacji - wtedy uznajemy go za słowo.

Rozwiązanie zadania 3.5.2

```
import re
def spr(x):
  if re.search("^[+-]?[0-9]+([.][0-9]+)?$", x):
    print(x, "jest liczbą")
  else:
    print(x, "NIE jest liczbą")
```

Rozwiązanie zadania 4.2.1

```
def zlicz(l):
    s = {}
    for e in l:
        s[e] = s.get(e, 0) + 1
    for k in s:
        print (str(k) + " wystepuje " + str(s[k]) + " razy")
```

Zwróć uwagę że:

- wykorzystujemy słownik s do trzymania mapy element - ilość powtórzeń
- przed pętlą zliczającą inicjujemy s jako pusty słownik
- w pętli zliczającej (iterującej po liście) używamy metody get słownika, aby pobrać wartość odpowiadającą danemu kluczowi lub zero gdy takiego klucza nie było, zamiast tej metody możliwyśmy użyć konstrukcji `if e in s:` do rozróżnienia przypadku pierwszego i kolejnego wystąpienie elementu e.
- po pętli zliczającej mamy osobną pętlę iterującą po słowniku celem wypisania ilości wystąpień

Rozwiązanie zadania 4.4.1

```
def wykonaj(lista, funkcja):
    for x in lista:
        funkcja(x)
```

Zwróć uwagę że funkcję przekazujemy do zmiennej tak samo jak dowolny inny argument (zmienną). Użycie funkcji przechowywanej w zmiennej polega na wywołaniu tej zmiennej z nawiasami okrągłymi i ewentualnymi argumentami tej funkcji.

Rozwiązanie zadania 4.4.2

Możemy zdefiniować słownik, w którym kluczem jest nazwa działania a wartością funkcja je realizująca. Zaleta takiego podejścia jest łatwe rozszerzania takiego kodu o nowe działania (poprzez wstawienie kolejnej pary do słownika, co może dziać się w trakcie pracy programu).

Rozwiązanie zadania 4.9.1

```
import sys, os, select

def czytaj(timeout):
    bufor = ""
    while True:
        rdfd, _, _ = select.select([sys.stdin], [], [], timeout)

        if not rdfd:
            return bufor

        for fd in rdfd:
            bufor += os.read(fd.fileno(), 1024).decode()

czytaj(13)
```

Zwróć uwagę że:

- funkcja w nieskończonej pętli wykonuje `select` z ustawioną wartością timeoutu
- w oparciu o wynik działania select funkcja rozróżnia przypadek timeoutu (rdfd jest `None`, czyli `not` rdfd jest prawdą) i zwraca w tej sytuacji wczytane wcześniej dane
- jeżeli nie było timeoutu, a pojawiły się jakieś dane (rdfd nie jest `None`) to funkcja wczytuje je z użyciem funkcji read
- jako że funkcja read wymaga określenia jakiegoś skończonej wielkości bufora, to do wczytywania danych użyta jest pętla for, co zapewnia wczytanie wszystkich dostępnych w danym momencie danych
- na wczytanych danych użyta jest metoda `decode` w celu zamiany ciągu bitowego na napis i tak przekonwertowane dane dodawane są do bufora napisowego (o dynamicznie dostosowywanej przez Pythona długości)

Rozwiązanie zadania 4.9.2

```
def zapisz(slownik, nazwa):
    plik = open(nazwa, 'wt', encoding='utf8')
    for klucz in slownik:
        plik.write(klucz + "\t" + slownik[klucz] +"\n")
    plik.close()
```

Zwróć uwagę że:

- funkcja użyta open z argumentami wskazującymi otwarcie pliku tekstowego w trybie do zapisu, w tym przykładzie podaliśmy także kodowanie, ale jest to zbędne gdyż podany utf8 bybły i tak użyty domyślnie
- następnie w ramach pętli przechodzącej po przekazanym do funkcji słowniku (dokładnie po kluczach w tym słowniku) wywołujemy funkcję write i podajemy do niej odpowiednio przygotowany napis (jako że plik otworzyliśmy w trybie tekstowym, write przyjmuje napisy)
- po wpisaniu wszystkich danych dokonujemy zamknięcia pliku (bez wykonania tej operacji część danych mogłaby nie być fizycznie zapisana w pliku w momencie kończenia funkcji)

Rozwiązanie zadania 5.1.1

```
import os, subprocess

pid = os.fork()

if pid == 0:
    res = subprocess.run(["ps", "-f"], stdout=subprocess.PIPE)
    print("Standardowe wyjście z komendy to: " + res.stdout.decode())
else:
    print("Mój PID to ", os.getpid(), "PID mojego potomka to:", pid)
```

Zwróć uwagę że:

- korzystamy z funkcji fork do rozdzielania procesu na dwa (rodzica i potomka)
- każdy z tych procesów zaczyna wykonywanie od wyjścia z funkcji fork, przy czym różni się w nich wartość, którą ta funkcja zwróciła
- wartości tej używamy do rozróżnienia rodzica od potomka
- w rodzicu (którym otrzymał niezerową wartość, będącą numerem PID utworzonego potomka) wypisujemy stosowny komunikat na temat numerów PID
- w potomku używamy `subprocess.run` do uruchomienia wskazanego polecenia w kolejnym potomku i przechwycenia jego wyjścia (opcjonalny argument `stdout=subprocess.PIPE`)
- po zakończeniu działania subprocess wypisujemy dane otrzymane na standardowym wyjściu uruchomionego polecenia
- dane te w ogólności są danymi binarnymi i Python używa typu ciągu bajtowego do ich przechowywania, my wiemy że ps wypisuje tekst na standardowym wyjściu zatem korzystamy z metody decode do jego konwersji na napis